DOCUMENT RESUME

ED 060 919 52 LI 003 610

AUTHOR	Silver, Steven S.; Meredith, Joseph C.
TITLE	DISCUS Interactive System Users' Manual. Final Report.
INSTITUTION	California Univ., Berkeley. Inst. of Library Research.
SPONS AGENCY	Office of Education (DHEW), Washington, D.C. Bureau of Research.
BUREAU NO	BR-7-1085
PUB DATE	Sep 71
GRANT	OEG-1-7-071085-4286
NOTE	173p.;(11 References)
EDRS PRICE	MF-$0.65 HC-$6.58
DESCRIPTORS	*Automation; Computer Assisted Instruction; Computer Programs; Data Bases; Electronic Data Processing; *Information Processing; *Information Retrieval; *Library Education; *Library Science; Man Machine Systems; Manuals; Programing Languages; Research
IDENTIFIERS	*University of California Berkeley

ABSTRACT

The results of the second 18 months (December 15, 1968-June 30, 1970) of effort toward developing an Information Processing Laboratory for research and education in library science is reported in six volumes. This volume contains: the basic on-line interchange, DISCUS operations, programming in DISCUS, concise DISCUS specifications, system author mode, and exercises. DISCUS is an interpretive man-computer interface system. The six parts of this manual contains: (1) an introduction to the general idea of computer assisted instruction, (2) an explanation of the several DISCUS statements, (3) a discussion of the role of the programmer vis-a-vis the author/instructor, (4) definitions and specifications, (5) a description of the program debugging facilities provided by the DISCUS language and (6) six series of exercises supplementing Parts II and III. (Other volumes of this report are available as LI 003607 through 003609, and LI 003611). (Author/NH)

ED 060919

U.S. DEPARTMENT OF HEALTH,
EDUCATION & WELFARE
OFFICE OF EDUCATION
THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL OFFICE OF EDUCATION POSITION OR POLICY.

FINAL REPORT
Project No. 7-1085
Grant No. OEG-1-7-071085-4286

DISCUS INTERACTIVE SYSTEM
USERS' MANUAL

By

Steven S. Silver
Joseph C. Meredith

Institute of Library Research
University of California
Berkeley, California 94720

September 1971

The research reported herein was performed pursuant to a grant with the Office of Education, U.S. Department of Health, Education, and Welfare. Contractors undertaking such projects under Government sponsorship are encouraged to express freely their professional judgment in the conduct of the project. Points of view or opinions stated do not, therefore, necessarily represent official Office of Education position or policy.

LI 003 610

U.S. DEPARTMENT OF
HEALTH, EDUCATION, AND WELFARE

Office of Education
Bureau of Research

TABLE OF CONTENTS

page

INTRODUCTION

Definition. 1
Background. 1
Documentation 4
Users' Manual Conventions 4
Organization. 5

PART I - THE BASIC ON-LINE INTERCHANGE

Language. 7
Approach. 8
The Nature of "Interaction" 8
Three DISCUS Tools. 9
The DISCUS System 16
Compilation and Execution 16
Operational Procedures. 18
Student Data Sets 19
Revision. 19

PART II - DISCUS OPERATIONS

Operation Codes 21
WRITE or W. 22
WRITE(NF) or W(NF). 30
WRITE(ND) or W(ND). 31
ANSWER or A 34
ANSWER(NF) or A(NF) 39
SCAN or SC. 40
Scanning for Words. 42
Scanning for Literals .44
Scanning for Words and Literals Intermixed. 45
Punctuation Marks and Special Characters. 46
Suboperands 47
Spoilers. 50
Expanding Contents of Variables into Other Strings. 53
Variables 55
DEFINE(A) or D(A) - DEFINE(C) or D(C) 56
SET or S. 57
TEST or T 62
The Decision Process. .67
MATCH and FAIL Counters 68
JUMP or J 69
Recapitulation. 69
The Block Structure (MATCH - FAIL - END). 70
BLOCK or B. 77
USE or U. 77
FRAME or FR 79
NOTE or N 84

TABLE OF CONTENTS (cont.)

page

PART III - PROGRAMMING IN DISCUS

Identifying the Programmer. 87
CAI as a Product. 88
Satisfactory Computer Assisted Dialog 88
Programming as an Exercise in Anonymity 89
Preparations. 90
Corpus. 91
Starred Variables 111

PART IV - CONCISE DISCUS SPECIFICATIONS

Specifications. 113
Architecture. 113
Requirements. 113
Current Implementation. 114
Glossary. 116
OPCODES 119
WRITE or W. 119
WRITE(NF) or W(NF). 119
WRITE(ND) or W(ND). 119
ANSWER or A 120
ANSWER(NF) or A(NF) 122
SCAN or SC. 123
DEFINE or D 126
SET or S. 127
TEST or T 130
JUMP or J 130
MATCH or M. 131
FAIL or F 131
BLOCK or B. 131
FRAME or FR 132
NOTE or N 132
END or E. 132
USE or U. 133
Job Control Language for Compiling DISCUS 134

PART V - SYSTEM AUTHOR MODE

Description 137
Distinguished from Proctor Author Mode. 137
Diagnostic Display. 138
System Author Mode Commands 138
EXIT (Berkeley) 140
END (UCLA). 140
EDITING 140

PART VI - EXERCISES. 141

FOREWORD

This report contains the results of the second 18 months (December 15, 1968 - June 30, 1970) of effort toward developing an Information Processing Laboratory for research and education in library science. The work was supported by a grant (OEG-1-7-071085-4286) from the Bureau of Research of the Office of Education, U.S. Department of Health, Education, and Welfare and also by the University of California. The principal investigator was M.E. Maron, Professor of Librarianship.

This report is being issued as six separate volumes by the Institute of Library Research, University of California, Berkeley. They are:

- Maron, M.E. and Don Sherman, et al. An Information Processing Laboratory for Education and Research in Library Science: Phase 2.

  Contents--Introduction and Overview; Problems of Library Science; Facility Development; Operational Experience.

- Mignon, Edmond and Irene L. Travis. LABSEARCH: ILR Associative Search System Terminal Users' Manual.

  Contents--Basic Operating Instructions; Commands; Scoring Measures of Association; Subject Authority List.

- Meredith, Joseph C. Reference Search System (REFSEARCH) Users' Manual.

  Contents--Rationale and Description; Definitions; Index and Coding Key; Retrieval Procedures; Examples.

- Silver, Steven S. and Joseph C. Meredith. DISCUS Interactive System Users' Manual.

  Contents--Basic On-Line Interchange; DISCUS Operations; Programming in DISCUS; Concise DISCUS Specifications; System Author Mode; Exercises.

- Smith, Stephen F. and William Harrelson. TMS: A Terminal Monitor System for Information Processing.

  Contents--Part I: Users' Guide - A Guide to Writing Programs for TMS
  Part II: Internals Guide - A Program Logic Manual for the Terminal Monitor System

- Aiyer, Arjun K. The CIMARON System: Modular Programs for the Organization and Search of Large Files.

  Contents--Data Base Selection; Entering Search Requests; Search Results; Record Retrieval Controls; Data Base Generation.

Because of the joint support provided by the File Organization Project (OEG-1-7-071083-5068) for the development of DISCUS and of TMS, the volumes concerned with these programs are included as part of the final report for both projects. Also, the CIMARON System, whose development was supported by the File Organization Project, has been incorporated into the Laboratory operation and therefore, in order to provide a balanced view of the total facility obtained, that volume is included as part of this Laboratory project report. (See Shoffner, R.M., et al., The Organization and Search of Bibliographic Records in On-Line Computer Systems: Project Summary.)

ACKNOWLEDGMENTS

Principal assistance in formulating the system specifications of DISCUS was provided by Allan Humphrey, Project Manager, Institute of Library Research. During the design and development phase, the project benefited from the active support of the staff of the Campus Computing Network, University of California at Los Angeles, and the use of its services and facilities.

During the entire period of testing and validating the system, and later in connection with the drafting of this manual, Rodney Randall, Systems Programmer, Institute of Library Research, participated very actively and effectively, contributing many hours of his personal time. He is solely responsible for the PILOT-to-DISCUS translator which automatically converts PILOT source coding to comparable DISCUS code.

In addition, we wish to thank and to commend the work of the Institute personnel who prepared these pages for publication:
Ellen Drapkin, Carole Fender, Bettye Geer, Linda Herold, Jan Kumataka, Barbara Parrish, and Rhozalyn Perkins.

INTRODUCTION

DEFINITION

DISCUS is an interpretive man-computer interface system, currently implemented as a conversational CAI language. It is programmed entirely in assembly language, for the IBM 360 series.

BACKGROUND

In July of 1967 the Institute of Library Research initiated Project No. 7-1085, *An Information Processing Laboratory for Education and Research in Library Science*, supported under Office of Education grant No. OEG-1-7-071085-4286, with contributory support by the University of California. In the design of such a laboratory, one of the important aspects to be investigated was the suitability of Computer Assisted Instruction (CAI) as a means of presenting certain types of library science materials to students in a graduate School of Librarianship. We needed to know what prior preparation of such materials would be required, what would be the programming problems in developing an on-line dialogue for instructional purposes, and how best to implement this kind of facility for graduate studies.

These requirements led to the actual writing and programming of a substantial amount of instructional material in the CAI medium, and to its implementation in the Information Processing Laboratory - using first teletype and typewriter terminals, and subsequently cathode ray tube (CRT) terminals acquired under a University grant for innovative projects in education.

When our research on the Information Processing Laboratory first began, we investigated the then existing languages of the type known as "selected character-string match languages," i.e., those capable of scanning free input for specified key elements, then acting on success or failure in finding these elements as instructional branching determinators. Among those considered, a new language under development in the Office of Information Systems, University of California at San Francisco, called PILOT*, appeared to be the most promising, and accordingly it was chosen as the new language in which we would encode our first courses of instruction. However, the choice was necessarily provisional, since PILOT itself was still under development, and there was no guarantee that it would stabilize in exactly the form which would be best for the system envisaged for the Information Processing Laboratory.

During 1968 and most of 1969, most of our CAI materials were programmed in PILOT, and were run under the PILOT system operating on an IBM 360/50 at the San Francisco campus Computing Center, with linkage by commercial grade telephone lines and acoustic couplers to our mechanical terminals in the Laboratory. This arrangement was oocasioned by the fact PILOT requires considerably more core memory than was available to us through the IBM 360/40 system serving the other needs of the Information Processing Laboratory, sited on the Berkeley campus. At the same time it demonstrated the feasibility of such an operation conducted at a remote distance from the central processing unit.

With the acquisition of the CRT system as the primary terminal hardware for the Laboratory, the need for CRT-compatible software became controlling. Since PILOT does not provide this kind of interface, it was necessary for us either to try to write one or to adapt another language which already incorporated this feature. The latter appeared to be the more feasible course, especially in view of the problem of core requirements raised by continued use of PILOT.

---

*Karpinski, R., et al, *PILOT....a conversational language - User Guide.* Office of Information Services, University of California Medical Center, San Francisco, California. 12/1/68.

Meanwhile, the Institute of Library Research at the University of California at Los Angeles had generated certain papers dealing with LYRIC, the CAI language developed by Gloria M. and Leonard C. Silvern.* In the fall of 1968, Steven S. Silver (Staff, Institute of Library Research, UCLA) undertook to examine the feasibility of adapting LYRIC to our needs. However, the problem of making the necessary changes proved more formidable than that of writing a new language from the beginning, and in January, 1969, it was decided that we should proceed on the latter basis. We were aware, of course, that there is much to be said in favor of standardization of CAI languages, but felt that in the context of the kind of research we were performing, a new departure from existing forms was justified. It now appears that much additional research and development work remains to be done before a complete spectrum of CAI language characteristics and capabilities will be available, and that standardization should be based on such a spectrum rather than on an attempt to make all programs look alike.

The features to be embodied in the new language were the subject of numerous exchanges between Institute staff at Berkeley and Los Angeles, and the version finally decided on was specified on March 24, 1969. These specifications followed the dicta that the system:

(1) accommodate natural language input

---

*Described in *Computer-assisted instruction: specifications for CAI programs and programmers,* by Gloria M. Silvern and Leonard C. Silvern. Proceedings of the 21st Annual Conference of the Association for Computing Machinery. ACM Publ. P-66 (Thompson Book Co., Washington, D.C., 1966) 1. 57-65.

Three of the papers referred to are limited distribution items. The fourth, *A Description of LYRIC, a language for remote instruction by computer*, by Steven S. Silver, appears as Appendix III in the final report on Project No. 7-1083, Grant No. OEG-1-7-071083-6068, *A Study of the Organization and Search of Bibliographic Holdings Records in On-line Computer Systems: Phase I*, by Jay Cunningham, Will Schieber, and Ralph Shoffner, Institute of Library Research, University of California, Berkeley, California, March, 1969.

(2) maintain individual student data

(3) restart individual students at the appropriate location, following a period of sign-off

(4) operate under a multi-course, multi-terminal time sharing system

(5) provide interface with CRT terminals

(6) use as little core memory as possible, both in compiling and in execution

Owing to the fact that a considerable amount of actual programming had been carried out prior to the final specification, it was possible to implement the new language for operational testing in May, 1969. In July, 1969, it was implemented at both the Berkeley and the UCLA campuses under the name of "DISCUS". Since that time, it has been undergoing continuous testing, evaluation, and revision. We now feel that it is sufficiently reliable and effective to justify its release for general use.

DOCUMENTATION

Complete documentation is available for prospective users of DISCUS, at cost, and with the understanding that suitable credits will be accorded to the Office of Education, Department of Health, Education, and Welfare for their support, and to the Institute of Library Research, University of California.

USERS' MANUAL CONVENTIONS

Since there are two kinds of users to be considered, one concerned with the programming and implementation of materials in the system and the other with actual consummation of dialogue at the terminal, we have adopted the following convention for the purposes of this manual:

By USER is meant the author, instructor, or coder using the DISCUS system to develop instructional or other dialogue materials.

By STUDENT is meant a person using a CRT terminal, interacting with the system. (It should be understood, however, that the system is not necessarily limited to educational uses.) Note also that the USER (i.e., author) must use the system as does a STUDENT in order to prepare and debug instructional programs.

A glossary of technical terms as they are used in this manual is provided in Part IV (CONCISE DISCUS SPECIFICATIONS).

Where necessary to draw attention to one or more blanks in examples given in the text where their presence might be overlooked, they are represented by the letter "b" with hyphen over-strike, thus:

ƀ = blank.

ORGANIZATION

It is not intended that this manual be read in strict page sequence. It should be read and studied in much the same way as that in which it was written - as an interweaving of needs and purposes, of explanation and speculation, of rules, examples, warnings, and invitations. One should not feel uneasy in exploring this material in seemingly random fashion, nor too exasperated when he finds it necessary to retrace earlier steps.

What we seek is a *construct* in the ancient sense of a piling up, a heaping together of elements which will in due course combine themselves in a manifest pattern. The functional relationships of the various pieces of the system cannot be well understood until something (not necessarily everything) is known about each. Until a concept of these relationships is achieved, the pieces themselves will have little meaning.

The manual is organized in six parts, as follows:

I. THE BASIC ON-LINE INTERCHANGE

This part is intended to introduce the general idea of CAI programming, and to demonstrate the operation of three of the standard DISCUS commands.

II. DISCUS OPERATIONS

An explanation of the several DISCUS statements, and a discussion of the decision process, block structures, and variables.

III. PROGRAMMING IN DISCUS

Discussion of the role of the programmer vis-a-vis the author/instructor. Technical considerations bearing on the design of CAI routines. Examples of useful subroutines. Advanced DISCUS programming.

IV. CONCISE DISCUS SPECIFICATIONS

Definitions and specifications.

V. SYSTEM AUTHOR MODE

Description of the program debugging facilities provided by the DISCUS language.

VI. EXERCISES

Six series of exercises supplementing Parts II and III.

PART I - THE BASIC ON-LINE INTERCHANGE

LANGUAGE

The kind of programming we will be dealing with is generally termed "high level," in that it rises atop a substructure of service routines prescribed in great detail, routines which we can rely upon without worrying about how they do the things they do. Theoretically, the highest level of programming language would be ordinary written communication, as if we were to tell the computer, in so many words, "I want you to program yourself to discuss counterpoint," or "How are you feeling today?" Of course at such a level, or any other level above that of detailed bit-manipulation, one does not really communicate with a computer, but with another person, one who has - we hope - foreseen at least part of our needs and has provided a program to accommodate them.

Unfortunately, the more elaborate the structure, the more costly becomes the effort of maintaining verbal or near-verbal communication through the computer. Simplicity at the top can mean ghastly complications near the bottom, all of which exact a price in terms of computer resources.

DISCUS tries to cut through some of these complications by dealing with the computer's operating system in quite fundamental terms rather than through the intermediation of one of the medium- or high-level languages such as FORTRAN, SNOBOL, or PL/1. This accounts for DISCUS' speed and economy, as well as for the fact that not everything is made simple and easy for the user (i.e., the programmer or encoder). In order to program properly in DISCUS - that is, to write conversational sequences of real versatility and power - the user must be adept with a number of highly specialized tools in various combinations, rather than with a number of all-purpose tools.

APPROACH

In trying to decide how to present a system which needs to be seen in its entirety in order to be perceived as a system at all, we have concluded that a very general approach will be well worth the risk of a few initial misconceptions which will be readily corrected in subsequent portions of the Manual.

THE NATURE OF "INTERACTION"

In order to establish an "interactive" situation, the programmer wants to force the computer to respond in a certain way to stimuli coming to it from some outside source - in our case from a CRT (cathode-ray tube) terminal. Basically the stimulus will always be a button-push, such as the user pressing (or thumping, if he likes) a key marked "attention," or "send," or "interrupt," or "carriage return". This act is like prodding a dumb animal with a stick. One expects a response of some kind unless the brute is very sick indeed. Usually the prod means "Hey, look what I wrote for you on my keyboard!"

The computer looks.

Its program tells it to react in a certain way to what it sees, depending on what that happens to be.

It reacts, usually by putting together some kind of message and flipping it to a slow-footed retainer for inscribing on the face of the cathode-ray tube screen at the terminal. If we converted microseconds to a more comprehensible scale, we might say that the computer handed the message to a stone-carver who only worked Tuesdays - but that would be all right, since the terminal user wouldn't be heard from again for about a year anyway.

It is customary to represent a dialogue between the computer and the individual as beginning with the computer, but this is inaccurate and can be quite misleading. The computer always has to be prodded, even before it will say "Sign in, please." Viewed in this light, every action of the computer is a <u>reaction</u>. The human is always the protagonist, even though at times he may feel quite otherwise.

The business of the CAI programmer is to equip the computer with adequate instructions to permit it to cope in some reasonable way with questions, commands, and statements expressed in ways that are human and therefore subtle, unpredictable, and messy.

THREE DISCUS TOOLS

In order to demonstrate the basic mechanism, we now provide the user with three basic tools with which he can simulate this relationship. With these tools he can even write a primitive DISCUS program:

A codeword -	ANSWER	meaning	*"At this point in the program the human types something"*
A codeword -	SCAN	meaning	*"Try to recognize, in the answer, something specified here"*
A codeword -	WRITE	meaning	*"Write on the CRT screen whatever is specified here"*

SCAN and WRITE always refer to something specified by the coder:

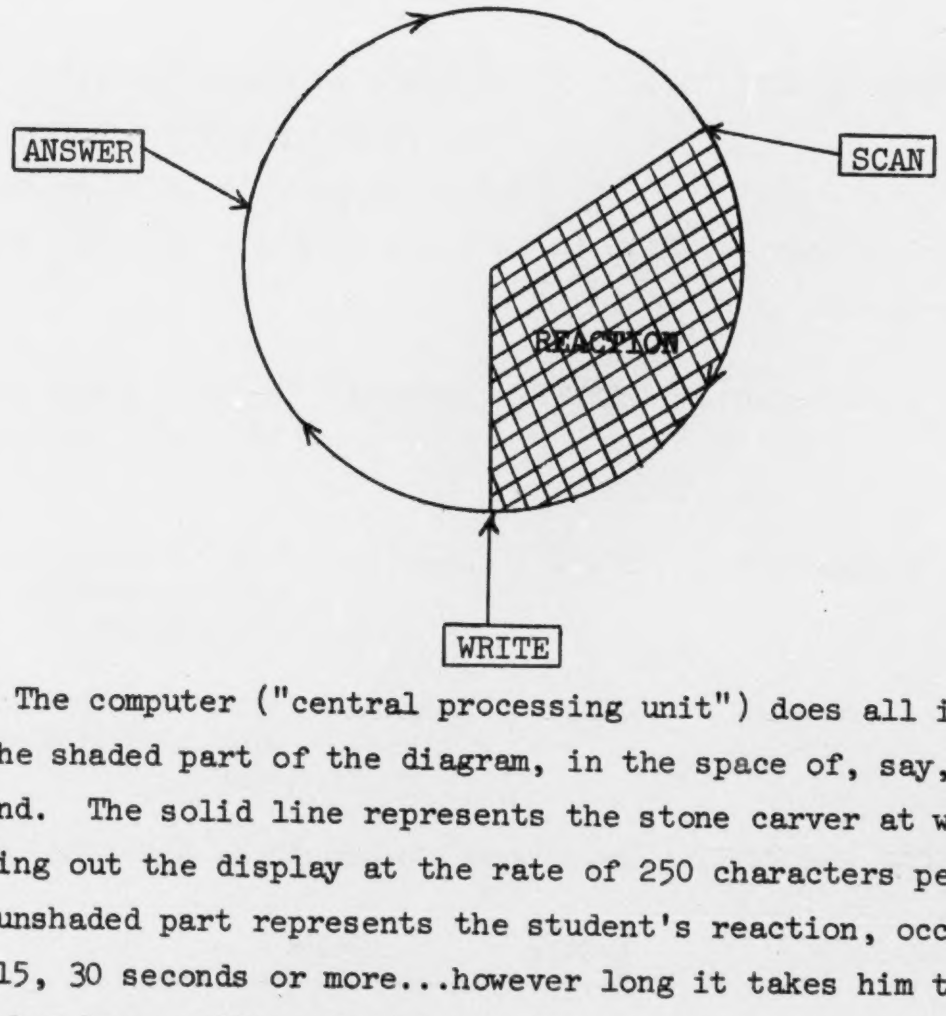
```
SCAN        GREEN          ;

WRITE       CORRECT        ;
```

ANSWER merely receives an unpredictable input typed by the student:

```
ANSWER       (?)          ;
```

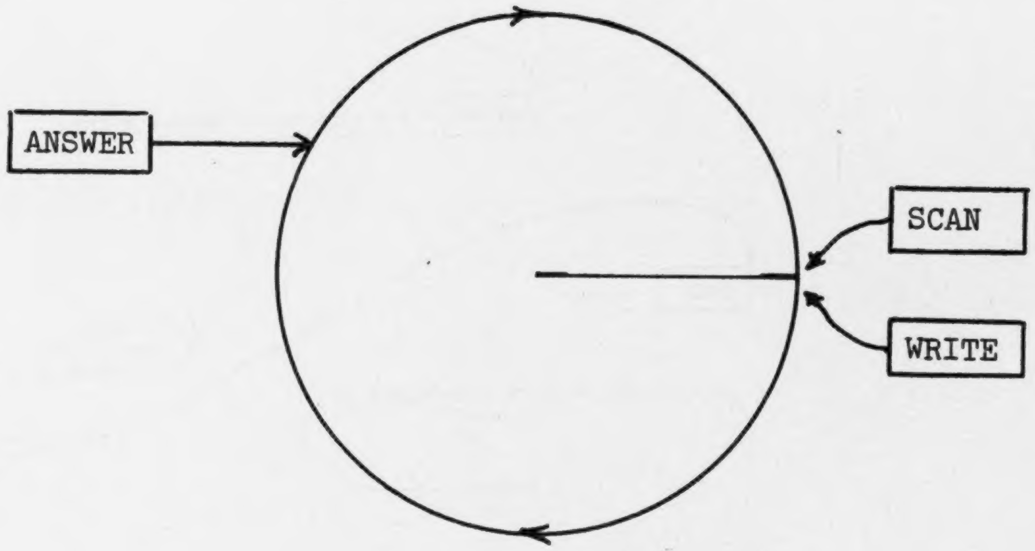
Observe the semicolons associated with each of the three codewords. They mark the end of that particular piece of coding, or statement. In other words, they "delimit the statement."

There are several additional codewords in the DISCUS system, but these three will suffice for the moment. Suppose we arrange them in a circle, to show the basic action-reaction cycle.

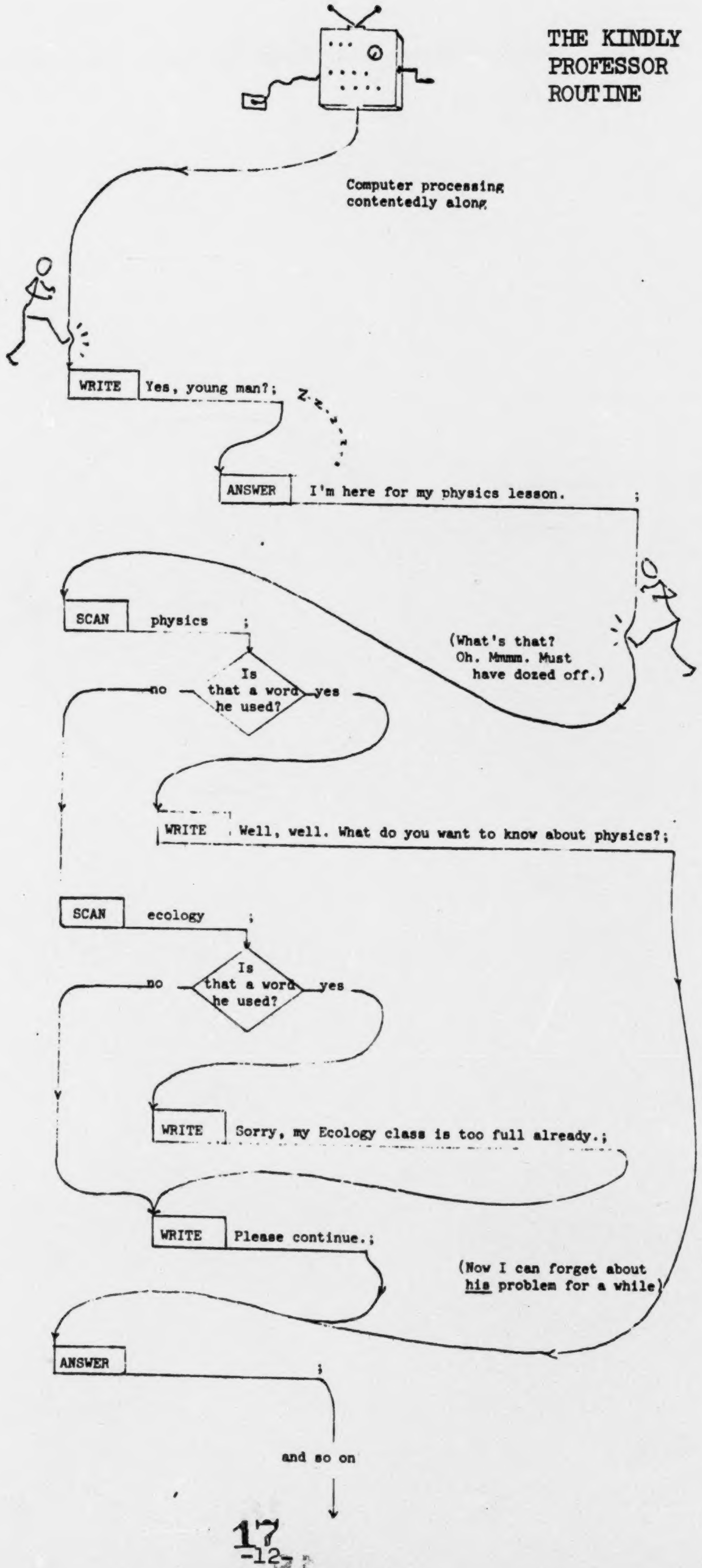


The computer ("central processing unit") does all its work in the shaded part of the diagram, in the space of, say, 1/1000 second. The solid line represents the stone carver at work, hacking out the display at the rate of 250 characters per second. The unshaded part represents the student's reaction, occupying 10, 15, 30 seconds or more...however long it takes him to read what has been written for him on the screen, to type in something new, and to push the "send" button.

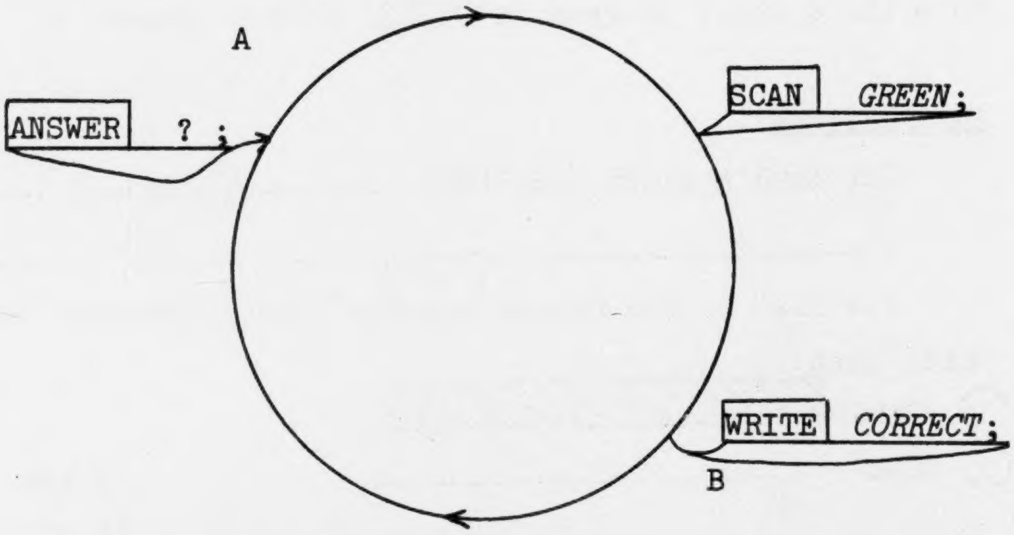
Proportioned according to time, the diagram would look like this:


ANSWER
SCAN
WRITE

THE KINDLY PROFESSOR ROUTINE

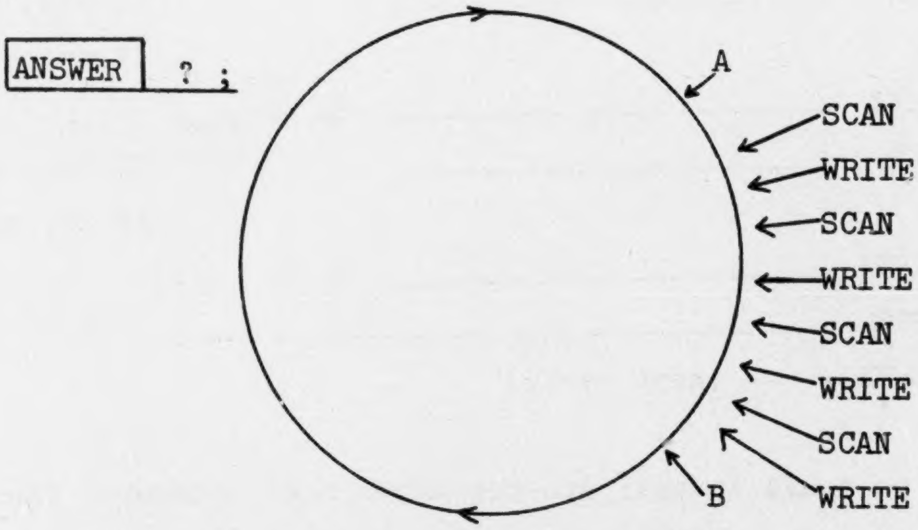

Computer processing contentedly along
WRITE
Yes, young man?;
Z z z z z
ANSWER
I'm here for my physics lesson. ;
SCAN
physics ;
(What's that? Oh. Mmmm. Must have dozed off.)
Is that a word he used?
no
yes
WRITE
Well, well. What do you want to know about physics?;
SCAN
ecology ;
Is that a word he used?
no
yes
WRITE
Sorry, my Ecology class is too full already.;
WRITE
Please continue.;
(Now I can forget about his problem for a while)
ANSWER
;
and so on

Another way to represent the process is with the coded statements themselves:


A
ANSWER ? ;
SCAN GREEN;
WRITE CORRECT;
B

Ⓐ represents the point where the "send" push or prod or is administered.
Ⓑ represents the point where the computer turns to other duties.

There is virtually no limit to the number of SCANs and WRITEs that can be programmed to follow a "send" signal:


ANSWER ? ;
A
SCAN
WRITE
SCAN
WRITE
SCAN
WRITE
SCAN
WRITE
B

Each of the four SCANs above could look for something different in the answer, and each of the WRITEs might be suppressed if the SCAN preceding it failed to find that something. In that little word "if" we become involved in the decision process, represented in the cartoon on page 12 by the inevitable diamond.

Below are some blank statements, arranged so that the decision process can be indicated by line number. The reader is invited to write a short program in DISCUS at this point.

Instructions:

Jot down a short question: ______________________________

______________________________?

Now fill in the blanks leaving line ① (student input) until last:

① ANSWER ▨;

② SCAN ____________________; If yes, go to ____; If no, go to ____;

③ WRITE ____________________; Go to ___;

④ SCAN ____________________; If yes, go to ____; If no, go to ____;

⑤ WRITE ____________________; Go to ___;

⑥ SCAN ____________________; If yes, go to ____; If no, go to ____;

⑦ WRITE ____________________; Go to ___;

⑧ SCAN ____________________; If yes, go to ____; If no, go to ____;

⑨ WRITE ____________________; Go go ___;

⑩ SCAN ____________________; If yes, go to ____; If no, go to ____;

⑪ WRITE ____________________; Go to ___;

⑫ WRITE ______________________________;

⑬ ANSWER------ (next cycle)

Later we shall investigate the additional codewords for accomplishing the items on the right hand side of the page, but for the presend purposes we can waive this requirement.

There is no way in which we can comment on your first piece of DISCUS programming, but it <u>might</u> look like something like this:

Question: (from a previous WRITE) *Where is Rome?*

```
①  ANSWER  ▨▨▨▨▨▨▨▨▨▨▨▨▨▨▨;
②  SCAN    Italy               ;             If yes, go to 3 ;
                                             If no, go to  4 ;
③  WRITE   Correct             ; Go to 12 ;
④  SCAN    Georgia             ;             If yes, go to 5 ;
                                             If no, go to  6 ;
⑤  WRITE   I mean the original ; Go to 1 ;
           Rome                ;
⑥  SCAN    Europe              ;             If yes, go to 7 ;
                                             If no, go to  8 ;
⑦  WRITE   What country in     ; Go to 1 ;
           Europe?             ;
⑧  SCAN    Don't know          ;             If yes, go to 9 ;
                                             If no, go to 10 ;
⑨  WRITE   You really should   ; Go to 1 ;
⑩  SCAN    Asia                ;             If yes, go to 11 ;
                                             If no, go to  12 ;
⑪  WRITE   Hardly              ; Go to 12 ;
⑫  WRITE   It's in Italy.  Where is Paris?;
⑬  ANSWER  -------------
```

Note that in three cases the program doubles back to the original ANSWER statement, to give the user a chance to try a different reply to the same question. The ability to perform such recursion is indispensable in CAI.

THE DISCUS SYSTEM

A "CAI system" comprises all the hardware and software dedicated to the specific purpose of computer-assisted instruction, plus certain hardware and software customarily shared with other systems in the same computer center. The computer itself is a prime example of shared hardware. The computer's own operating system (in this case IBM's OS/360) is a good example of shared software.

Peripheral equipment, such as disc storage units, may be shared (i.e., their capacity allocated either on a physical location basis or on a real time basis) or they may be dedicated to a single system or use.

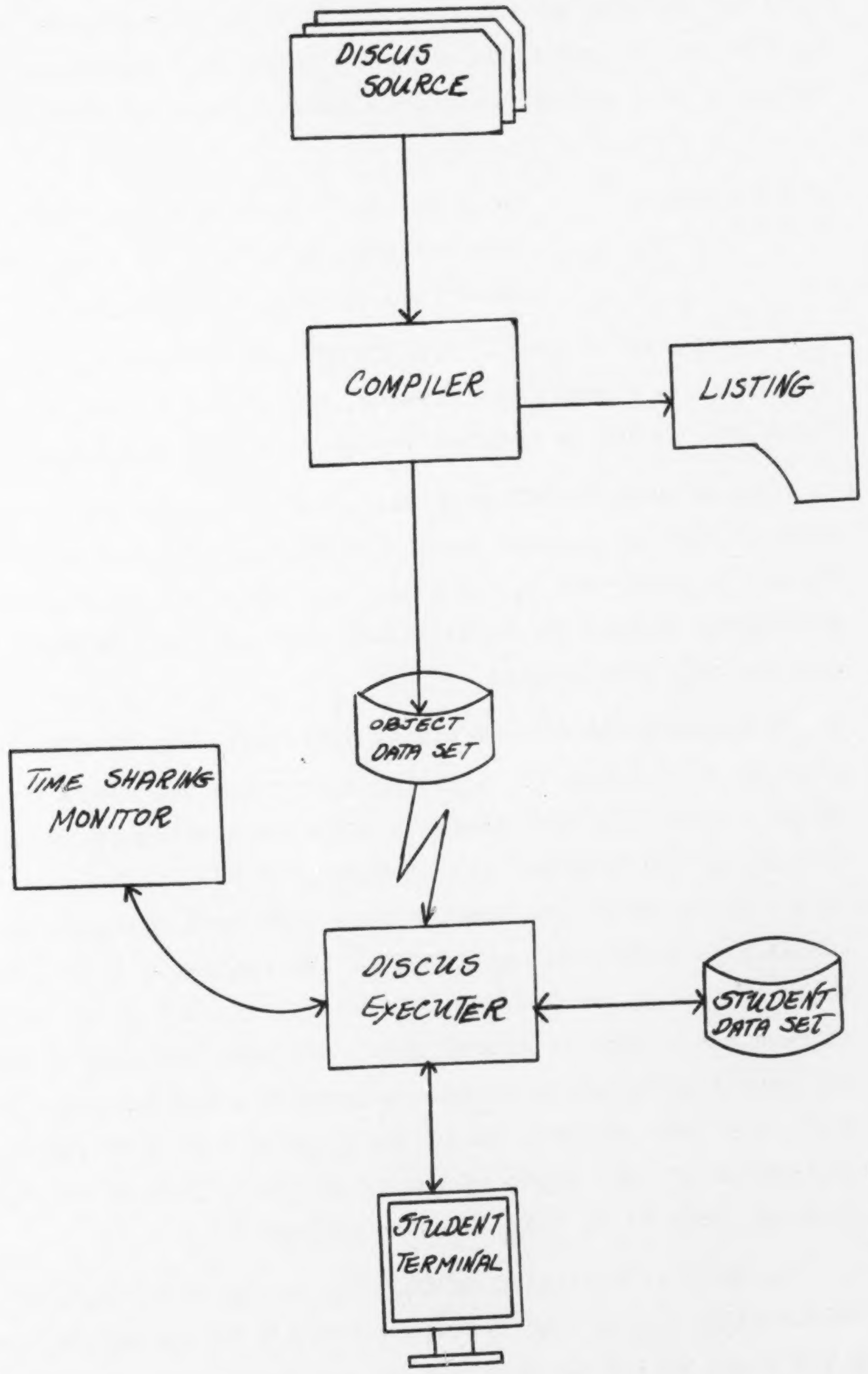
The services of control and auxiliary software, such as a time-sharing monitor, may also be shared between systems.

Although DISCUS can be spoken of as a CAI "language," it should be thought of more as the dedicated software components of a *CAI system*. Only after it has been implemented in the hardware, and actually "resides" in a computer, is the system complete and ready to operate.

COMPILATION AND EXECUTION

The system must be capable of two distinct and separate operations: *First*, it must be able to accept programs submitted to it for compilation into executable form, and to compile them - if in fact they are compilable according to the logic of the compiler. Normally an error in the source program submitted for compilation will not prevent compilation of the remainder of the program; only the faulty statement (and perhaps its associated statements) will be unexecutable.

Second, it must be able to execute programs, once they have been accepted, compiled, and stored in its repertory. "Execution" takes place when a student at a remote terminal is actually on line. Execution does not change the system itself; it is simply a product of the system according to the instantaneous conditions


DISCUS SOURCE
COMPILER
LISTING
OBJECT DATA SET
TIME SHARING MONITOR
DISCUS EXECUTER
STUDENT DATA SET
STUDENT TERMINAL

existing within it. Execution is time-related, in the same way that the running of a movie film is time-related - the product in the latter case being a static image on a screen, at a single instant, or a moving image in a span of several instants.

OPERATIONAL PROCEDURES

Four distinct operations are involved in the realization of the two basic functions described on the preceding page:

1. Two blocks of code - the DISCUS COMPILER and the DISCUS EXECUTOR - are read, assimilated, and stored by the computer. These two blocks or modules are shown in the adjoining diagram.

2. One or more SOURCE programs, consisting of data either in the form of tape or punched cards encoded according to the DISCUS rules, is submitted to the computer. This can be done at any time - minutes or months after the DISCUS COMPILER and EXECUTOR have been successfully established.

3. Whenever a SOURCE program is submitted, the DISCUS COMPILER attempts to compile it; that is, to arrange and store it in executable form. It also causes a complete listing of the compiled version of the program, called DISCUS OBJECT, to be printed. This listing shows the numbers that have been assigned by the COMPILER to individual statements, the condition code levels at which they will be presumed to operate, and the place in disc storage where each is stored away. It also includes a list of the labels attached to certain statements, showing where and how they have been referred to in the program. It also furnishes an indication of some types of coding errors. (A page of object listing, reduced to 45%, is shown on page 20.)

4. The program submitted as SOURCE, having been compiled to produce OBJECT code, can now be executed, subject to any malfunctions which might be encountered due to the above mentioned errors. A student activates a remote console, signs in, calls up the program by name or number, and he is off and running. This is the EXECUTION

phase. In a time-sharing system, execution can be going on in several different parts of the OBJECT MODULE simultaneously, or at least switching back and forth so rapidly that it seems simultaneous to several individuals using the system at the same time.

STUDENT DATA SETS

For each terminal in use, a portion of the computer's disc storage must be set aside for keeping a record of where execution is at any particular moment, for that particular terminal. It must also contain records pertaining to that user's entire terminal session: various scores and tallies, saved responses, etc. This reserved section is called a "student data set."

If the system permits users to sign off and to sign back on at a later time without having to begin all over again, it must keep a record of the "restart" location, plus all the scores and tallies left over from the previous session. Under this arrangement the student data set is stored away on disc during the time he is away from the terminal.

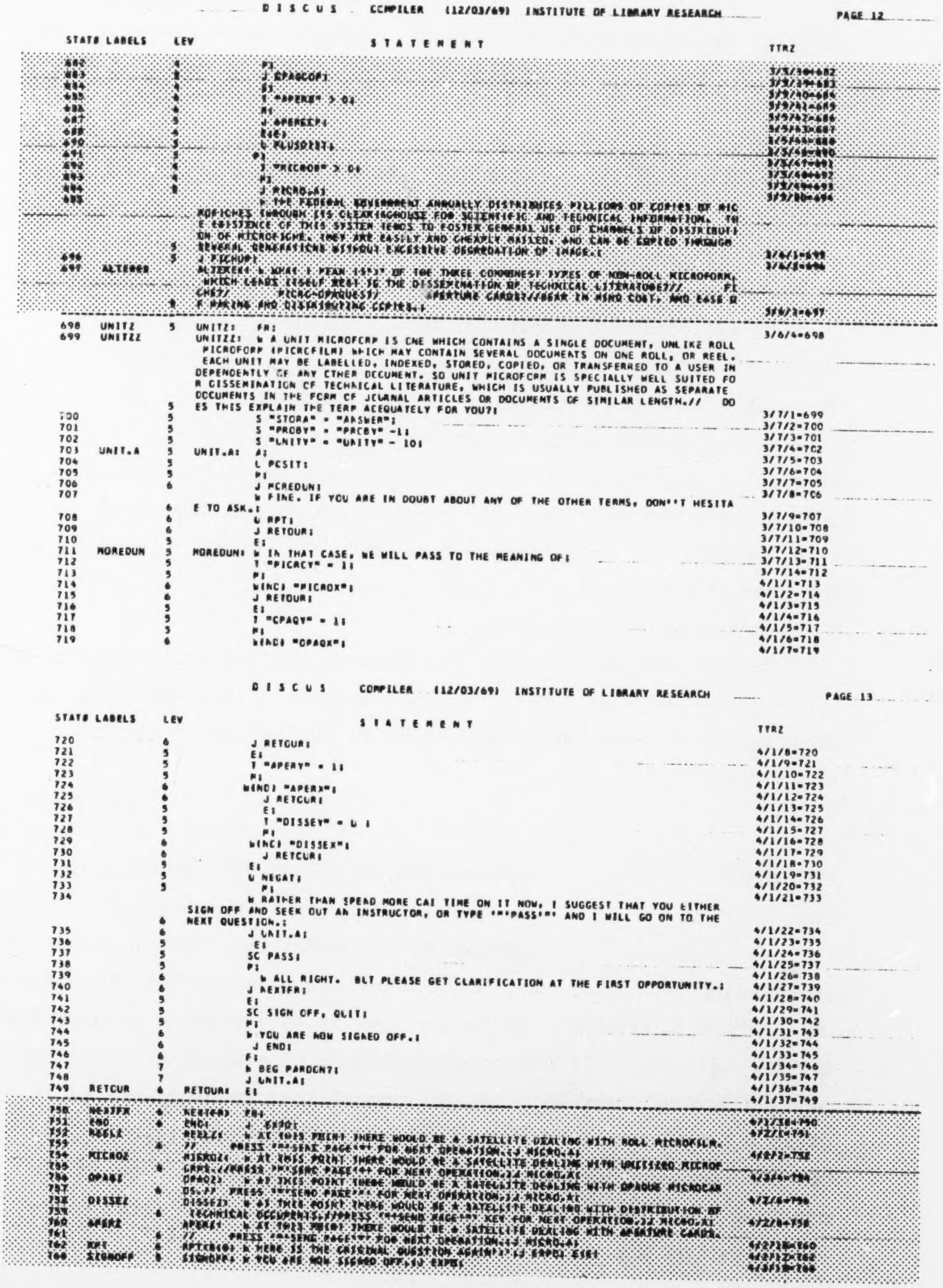
The general arrangements for "start" and "restart" are touched upon in Part IV, but the actual implementation will vary from installation to installation according to the design of the local monitor system.

REVISION

An existing OBJECT MODULE can be revised, or "updated," by submitting a revised SOURCE program to the COMPILER. In the course of recompiling this code, the COMPILER will obliterate the pre-existing OBJECT MODULE, unless a different area of disc storage is used for the new material. If the original SOURCE is available to the system on tape or disc, revision can be accomplished by using one of the utility programs provided by the IBM 360 Operating System. This permits changing certain card images recorded on the tape or disc, provided they have been numbered sequentially in the first place, then ordering the revised tape to be read (as SOURCE) into the system.

OBJECT CODE LISTING (Reduced)

Lines 698-749 = one small frame.



```
          D I S C U S   CCMPILER  (12/03/69)  INSTITUTE OF LIBRARY RESEARCH            PAGE 12

STAT# LABELS   LEV                    S T A T E M E N T                                     TTRZ

698   UNITZ     5   UNITZ:   FR:
699   UNITZZ        UNITZZ:  W A UNIT MICROFCRP IS CNE WHICH CONTAINS A SINGLE DOCUMENT, UNLIKE ROLL   3/6/4=698
                     PICROFORP (PICRCFILM) WHICH MAY CONTAIN SEVERAL DOCUMENTS ON ONE ROLL, OR REEL.
                     EACH UNIT MAY BE LABELLED, INDEXED, STORED, COPIED, OR TRANSFERRED TO A USER IN
                    DEPENDENTLY CF ANY CTHER DCCUMENT. SO UNIT PICROFCRM IS SPECIALLY WELL SUITED FO
                    R CISSEMINATICN CF TECHNICAL LITERATURE, WHICH IS USUALLY PUBLISHED AS SEPARATE
                    DOCUMENTS IN THE FCRM CF JCURNAL ARTICLES OR DOCUMENTS CF SIMILAR LENGTH.//   DO
                5   ES THIS EXPLAIN THE TERP ACEQUATELY FOR YOU?:
700             5            S "STORA" = "ANSWER";                                               3/7/1=699
701             5            S "PROBY" = "PRCBY" -1;                                             3/7/2=700
702             5            S "UNITY" = "UNITY" - 10;                                           3/7/3=701
703   UNIT.A    5   UNIT.A:  A;                                                                  3/7/4=7C2
704             5            L PCSIT;                                                            3/7/5=703
705             5            P;                                                                  3/7/6=704
706             6            J PCREDUN;                                                          3/7/7=705
707                          W FINE. IF YOU ARE IN DOUBT ABOUT ANY OF THE OTHER TERMS, DON''T HESITA 3/7/8=7C6
                6   E TO ASK.:
708             6            U RPT;                                                              3/7/9=707
709             6            J RETOUR;                                                           3/7/10=708
710             5            E;                                                                  3/7/11=709
711   MOREDUN   5   MOREDUN: W IN THAT CASE, WE WILL PASS TO THE MEANING OF:                     3/7/12=710
712             5            T "PICRCY" = 1;                                                     3/7/13=711
713             5            P;                                                                  3/7/14=712
714             6            WINC) "PICROX";                                                     4/1/1=713
715             6            J RETOUR;                                                           4/1/2=714
716             5            E;                                                                  4/1/3=715
717             5            T "CPAQY" = 1;                                                      4/1/4=716
718             5            P;                                                                  4/1/5=717
719             6            WINC) "OPAQX";                                                      4/1/6=718
                                                                                                 4/1/7=719

          D I S C U S    COMPILER  (12/03/69)  INSTITUTE OF LIBRARY RESEARCH            PAGE 13

STAT# LABELS   LEV                    S T A T E M E N T                                     TTRZ
720             6            J RETCUR;
721             5            E;                                                                  4/1/8=720
722             5            T "APERY" = 1;                                                      4/1/9=721
723             5            P;                                                                  4/1/10=722
724             6           WINC) "APERX";                                                       4/1/11=723
725             6              J RETCUR;                                                         4/1/12=724
726             5              E;                                                                4/1/13=725
727             5              T "DISSEY" = U ;                                                  4/1/14=726
728             5              P;                                                                4/1/15=727
729             6            WINC) "DISSEX";                                                     4/1/16=728
730             6              J RETCUR;                                                         4/1/17=729
731             5            E;                                                                  4/1/18=730
732             5            U NEGAT;                                                            4/1/19=731
733             5              P;                                                                4/1/20=732
734                          W RATHER THAN SPEND MORE CAI TIME ON IT NOW, I SUGGEST THAT YOU EITHER 4/1/21=733
                    SIGN OFF AND SEEK OUT AN INSTRUCTOR, OR TYPE '"'PASS'"' AND I WILL GO ON TO THE
                6   NEXT QUESTION.:
735             6            J UNIT.A;                                                           4/1/22=734
736             5             E;                                                                 4/1/23=735
737             5            SC PASS;                                                            4/1/24=736
738             5            P;                                                                  4/1/25=737
739             6              W ALL RIGHT.  BLT PLEASE GET CLARIFICATION AT THE FIRST OPPORTUNITY.: 4/1/26=738
740             6            J NEXTFR;                                                           4/1/27=739
741             5            E;                                                                  4/1/28=740
742             5            SC SIGN OFF, QUIT;                                                  4/1/29=741
743             5            P;                                                                  4/1/30=742
744             6            W YOU ARE NOW SIGNED OFF.:                                          4/1/31=743
745             6             J END;                                                             4/1/32=744
746             6            F;                                                                  4/1/33=745
747             7            W BEG PARDCN?:                                                      4/1/34=746
748             7            J UNIT.A;                                                           4/1/35=747
749   RETCUR    6   RETOUR:  E;                                                                  4/1/36=748
                                                                                                 4/1/37=749
```

PART II - DISCUS OPERATIONS

OPERATION CODES — A list of the several DISCUS OPCODES which govern execution of program statements follows:

Long form	Short form	See page
WRITE	W	22
WRITE (NF)	W(NF)	30
WRITE (ND)	W(ND)	31
ANSWER	A	34
ANSWER (NF)	A(NF)	39
SCAN	SC	40
DEFINE (A)	D(A)	56
DEFINE (C)	D(C)	56
SET	S	57
TEST	T	62
MATCH	M	68
FAIL	F	68
END	E	68
JUMP	J	69
BLOCK	B	77
USE	U	77
FRAME	FR	79
NOTE	N	84

There is no difference in operation between the long and short forms of OPCODE. Until one has become thoroughly habituated to working with DISCUS, the long form is recommended, because it makes printed listing of compiled programs somewhat easier to read.

The DISCUS compiler automatically assumes that any of the above forms is in fact an OPCODE if it is preceded by

a colon (marking the end of a preceding DISCUS label), with any number of intervening blanks;

a semicolon (marking the end of a preceding DISCUS statement), with any number of intervening blanks;

nothing at all (i.e., the beginning of the program);

and is followed by a blank. No special punctuation or other character is required to indicate its status as an OPCODE, and there is no danger of a properly positioned OPCODE being interpreted as a label,or as text to be displayed:

WRITE RIGHT;

will cause

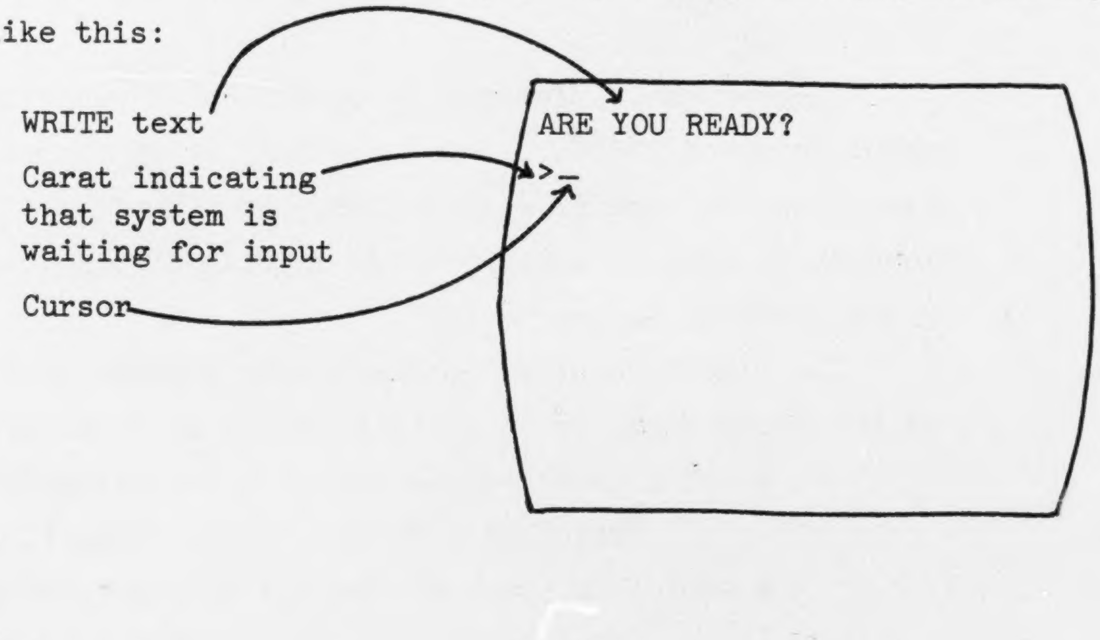
RIGHT

to appear on the CRT screen, provided the last non-blank character preceding "WRITE" (if any) is a delimiter (i.e., either a colon or a semicolon.)

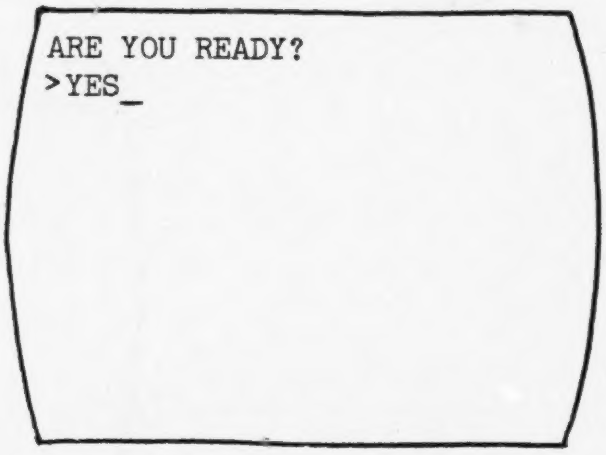
WRITE or W

A CRT screen can be "written" in one of two ways: through execution of a programmed WRITE statement by the computer, or by a student inputting characters at his terminal's keyboard. A typical display will consist of a block of programmed text, followed by an arrow or carat indicating the starting position of keyboard input to come, and a cursor to indicate the position in which the next keyed character will appear.

Before the terminal-user types any characters, the screen may look like this:



After the terminal-user types characters, but before he presses the "send" button, the screen might look like this:



```
ARE YOU READY?
>YES_
```

There are three forms of WRITE commands governing WRITE statements. Their form is independent of the purpose of the text to be written: any one of them can be used for conveying didactic text, for posing questions, or for responding to terminal input. Not all WRITE statements coded by the programmer are actually displayed in a particular terminal session. They are used selectively, depending entirely on the path which execution takes on that occasion.

A simple WRITE or W writes the screen from the top, after erasing all previous display material. It continues until the end of the statement, or until the end of the screen is reached, whichever happens first. In the latter case, the overflow is saved until the terminal user presses his "send"* button. This action is treated as an impromptu WRITE command, the screen is erased, and the remainder of the WRITE statement's operand (i.e., the text subject to that WRITE command) is displayed.

End-of-line formatting is automatically performed; that is, no word will be started that can't be finished on that same line.

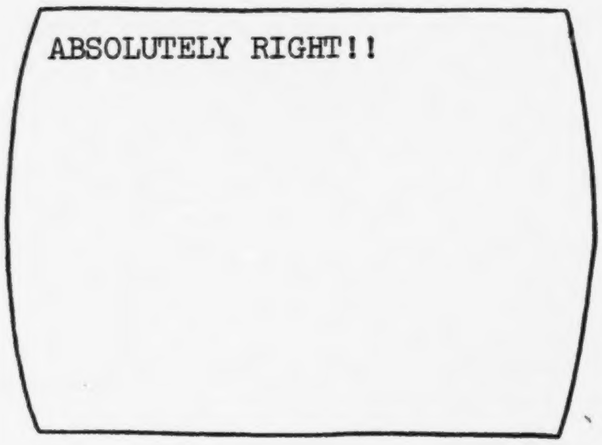
---

*Or other designated signal, e.g., "interrupt" or "attention," depending on the kind of terminal in use.

Examples

<u>Statement</u> - *WRITE ABSOLUTELY RIGHT!!;*

<u>Result</u>

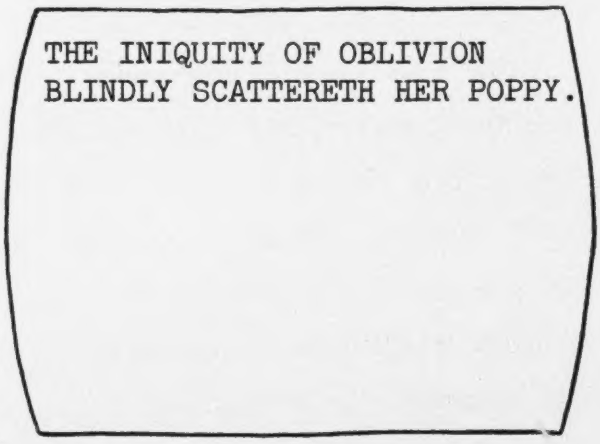


```
ABSOLUTELY RIGHT!!
```

<u>Statement</u> - *WRITE THE INIQUITY OF OBLIVION BLINDLY SCATTERETH HER POPPY.;*

(Throughout this manual we use an "example screen" with a line-width of 27 characters.)

<u>Result</u> -

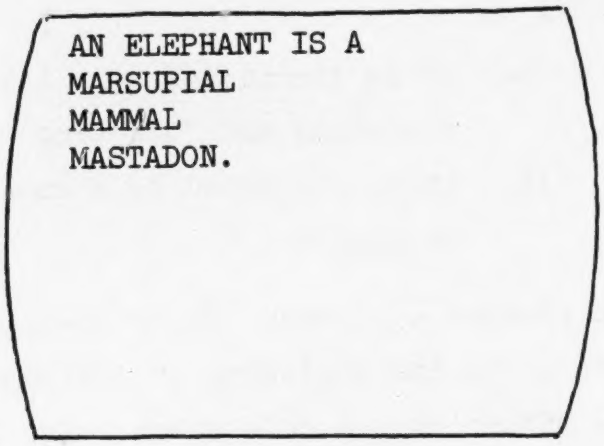


```
THE INIQUITY OF OBLIVION
BLINDLY SCATTERETH HER POPPY.
```

A line break may be <u>forced</u> at any point, inside or outside of a word, by inserting a slash (/) in the text. The slash is not displayed. Thus:

Statement - *WRITE AN ELEPHANT IS A/MARSUPIAL/MAMMAL/MASTADON.;*

Result -

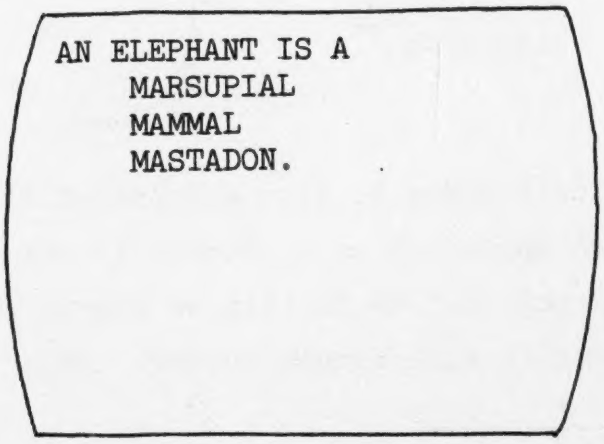


```
AN ELEPHANT IS A
MARSUPIAL
MAMMAL
MASTADON.
```

Formatting within the line can be accomplished by inserting blanks immediately after the slash:

Statement - *WRITE AN ELEPHANT IS A/ƀƀƀƀMARSUPIAL/ ƀƀƀƀMAMMAL/ƀƀƀƀMASTADON.;*

Result -



```
AN ELEPHANT IS A
    MARSUPIAL
    MAMMAL
    MASTADON.
```

(Note: the ƀ symbol is used only for explication herein, when necessary to emphasize the presence of blanks. It is never encoded as such. In keypunching, the space bar actually specifies blanks, in WRITE text.)

The slash is a "reserved character" in DISCUS. In a WRITE statement it always means "go to the beginning of the next line," <u>unless</u>

(a) it is identified as a literal* by single quotes preceding and following it, or

(b) it is contained in a quoted variable (discussed on page 55).**

Two slashes (//) mean "go to the beginning of the next line, then go to the beginning of the next line after that," having the effect of a double space. Any number of blank lines may be created in this manner.

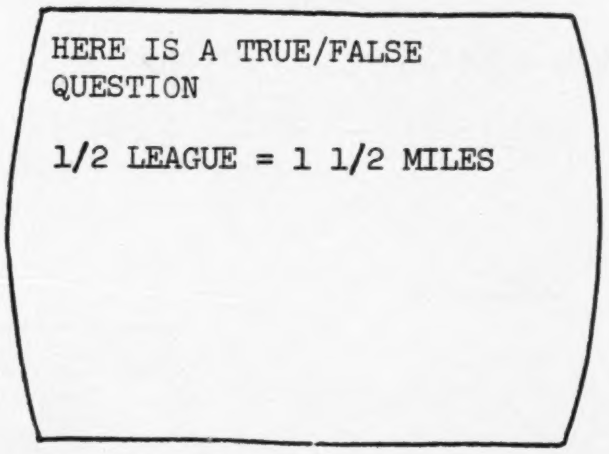
Statement - *WRITE HERE IS A TRUE'/'FALSE QUESTION//1'/ '2 LEAGUE = 1 1'/'2 MILES;*

Result -

Automatic line break

Forced line break, and double space

Delimiting semicolon is not displayed



```
HERE IS A TRUE/FALSE
QUESTION

1/2 LEAGUE = 1 1/2 MILES
```

The single quote is also a reserved character. Like the slash, it must be specified as a literal if one desires that it be written on the screen, but to do this we simply double it, rather than surrounding it with single quotes: Thus ''   , not '''   .***

---

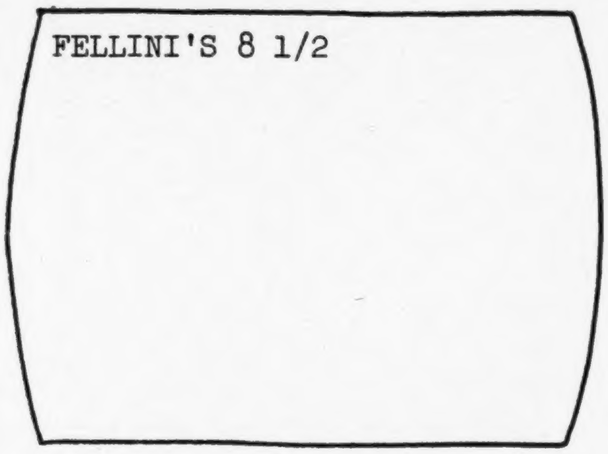
*Definition given on p.117.

**For the moment, these exceptions need not concern the reader.

***The DISCUS COMPILER always looks for pairs of single quotes. A good way of checking coding is to make sure that the total number of single quotes in a series of statements is an even number. The compiler will try to turn whole pages of source programming into a literal, following an odd-numbered single quote.

Statement - *WRITE FELLINI''S 8 1'/'2;*

Result



```
FELLINI'S 8 1/2
```

Three other reserved characters need to be considered:

: - the colon, which normally acts as a label delimiter

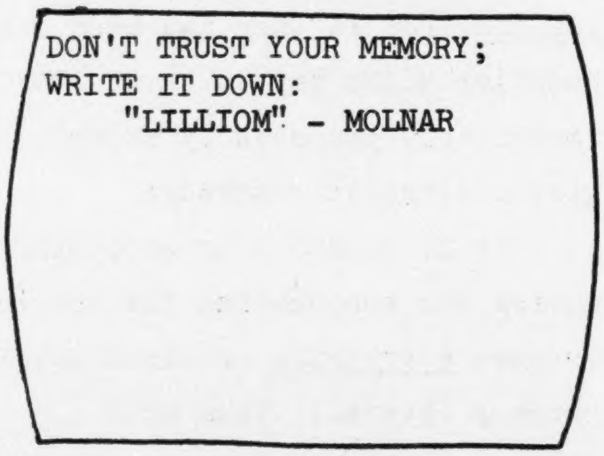
; - the semicolon, " " " " " statement "

" - the double quote, which normally surrounds the label of a variable whose contents are to be used at that point.

As with the slash, their special quality is suppressed by surrounding them with single quotes; they will then be displayed as literals.

Statement - *WRITE DON''T TRUST YOUR MEMORY';' WRITE IT DOWN':'/ƀƀƀƀ'"'LILLIOM'"' - MOLNAR;*

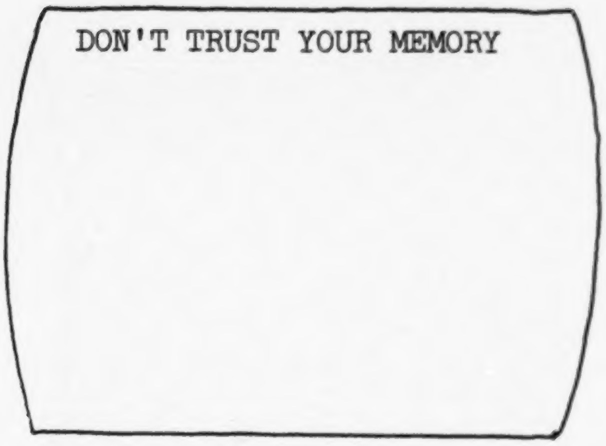
Result -



```
DON'T TRUST YOUR MEMORY;
WRITE IT DOWN:
    "LILLIOM" - MOLNAR
```

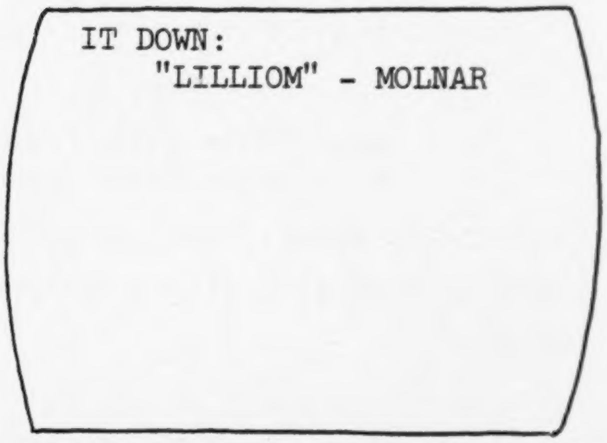
If, in the example immediately preceding, the single quotes had been omitted from around the semicolon, it would have been treated by the DISCUS COMPILER as a statement delimiter:

Result -



```
DON'T TRUST YOUR MEMORY
```

Followed by -



```
IT DOWN:
     "LILLIOM" - MOLNAR
```

If the single quotes had been omitted from around the double quotes, the compiler would have searched fruitlessly for a variable called LILLIOM. (Or if they had been omitted from around the colon, the compiler might try to treat DOWN as a label - but since DOWN is not immediately preceded by an active semicolon, the compiler would give a diagnostic message.

It is possible to economize somewhat in the use of single quotes for suppressing the special nature of reserved characters, because everything enclosed within a pair of single quotes becomes a literal. Thus both

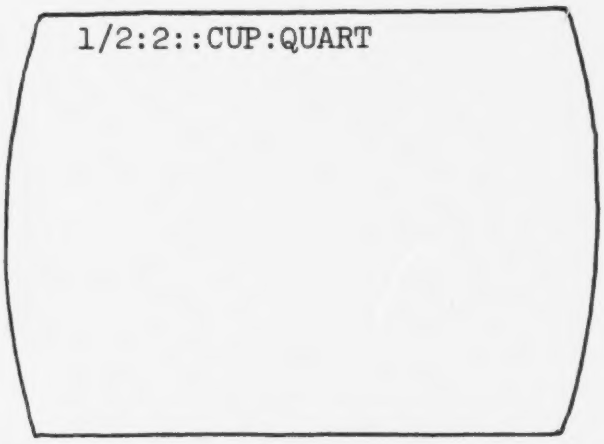
Statement - *WRITE 1'/'2':'2'::'CUP':'QUART*

and

Statement - *WRITE 1'/2:2::CUP:'QUART;*

have the same

Result -



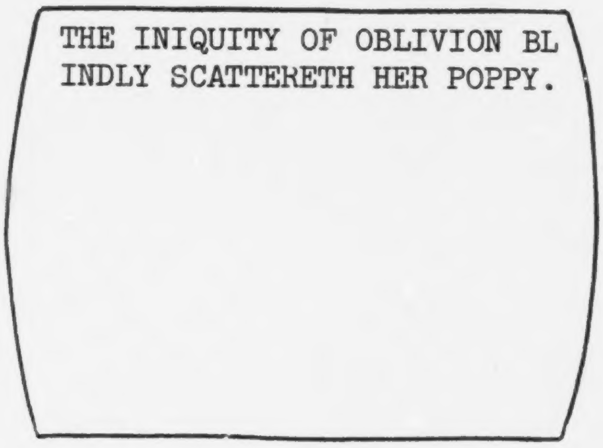
```
1/2:2::CUP:QUART
```

Normal end-of-line formatting is not affected by the above device, even if a line break occurs within the string surrounded by single quotes.

WRITE(NF) or W(NF) — This OPCODE causes the operand which follows it to be displayed <u>without</u> end-of-line formatting.

Statement - *WRITE(NF) THE INIQUITY OF OBLIVION BLINDLY SCATTERETH HER POPPY.;*

Result -



```
THE INIQUITY OF OBLIVION BL
INDLY SCATTERETH HER POPPY.
```

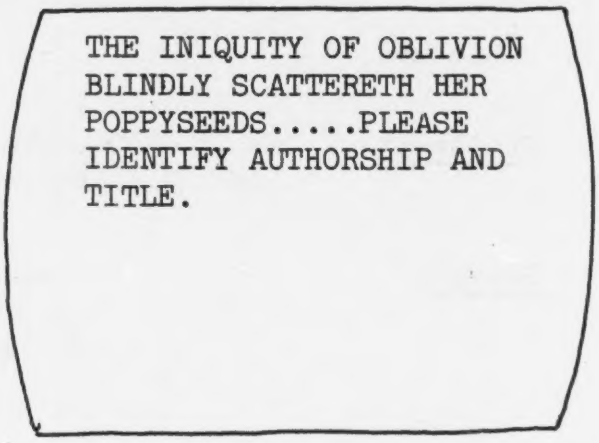
All other conditions are the same as with plain WRITE.

WRITE(ND) or W(ND) This OPCODE causes its operand to be displayed immediately following the preceding WRITE operand, without first erasing the screen. If the last preceding write statement has carried a WRITE opcode, end-of-line formatting will continue.

<u>Statements</u> - *WRITE THE INIQUITY OF OBLIVION BLINDLY SCATTERETH HER POPPYSEEDS.;*

WRITE(ND)ƀ....PLEASE IDENTIFY AUTHORSHIP AND TITLE.;

<u>Result</u> -



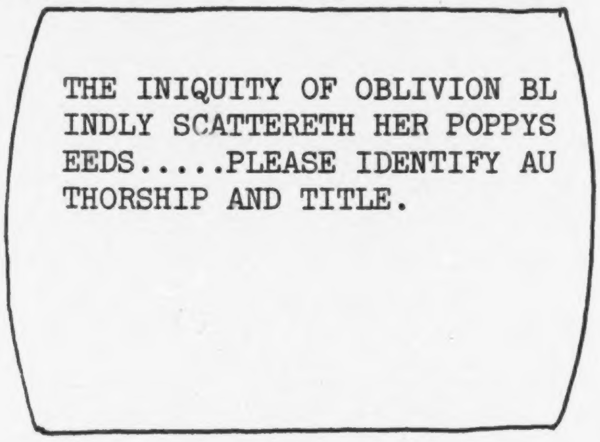
```
THE INIQUITY OF OBLIVION
BLINDLY SCATTERETH HER
POPPYSEEDS.....PLEASE
IDENTIFY AUTHORSHIP AND
TITLE.
```

If the last preceding write statement used a WRITE(NF) opcode, the WRITE(ND) result will be unformatted.

<u>Statements</u> - *WRITE(NF) THE INIQUITY OF OBLIVION BLINDLY SCATTERETH HER POPPYSEEDS.;*

WRITE(ND) ....PLEASE IDENTIFY AUTHORSHIP AND TITLE.;

<u>Result</u> -



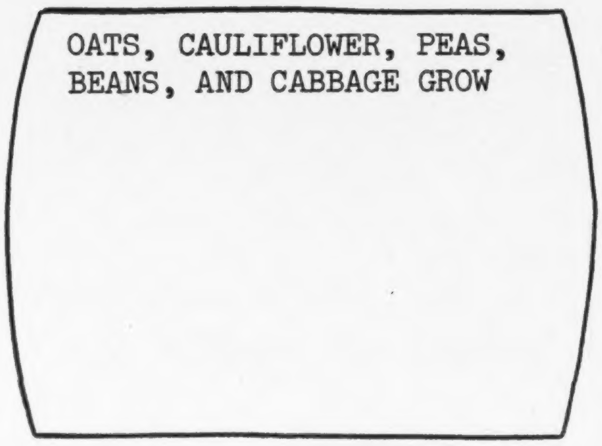
```
THE INIQUITY OF OBLIVION BL
INDLY SCATTERETH HER POPPYS
EEDS.....PLEASE IDENTIFY AU
THORSHIP AND TITLE.
```

The WRITE(ND) OPCODE is especially useful for displaying blocks of text simultaneously in smooth, consecutive format.

It can also be of service to the coder who undertakes the revision of a long statement already keypunched. Suppose an author or coder decides to remove a sentence from the middle of a 1000-character WRITE statement. If only the latter part of the statement is repunched, an unsightly gap in the text would occur, unless the sentence happened to be precisely 80 characters in length (which would allow him simply to remove the one card.) His alternative, afforded by the WRITE(ND) opcode, is to terminate the statement at the cut and turn the remainder into a WRITE(ND) statement. In miniature:

Statement - *WRITE OATS, CAULIFLOWER, PEAS, BEANS, AND CABBAGE GROW;*

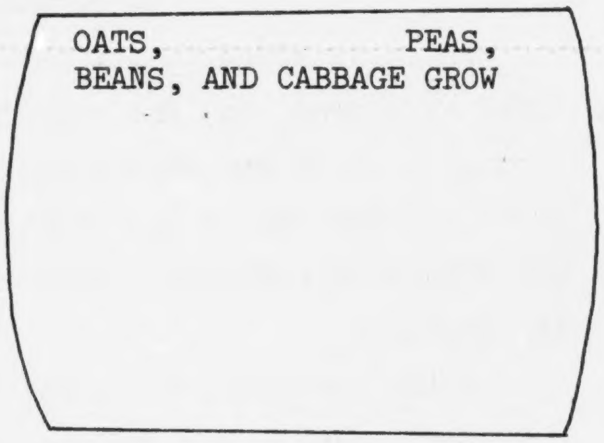
Result -



```
OATS, CAULIFLOWER, PEAS,
BEANS, AND CABBAGE GROW
```

(Example continued on next page)

If we were simply to substitute blanks for CAULIFLOWER, the display would look like this:

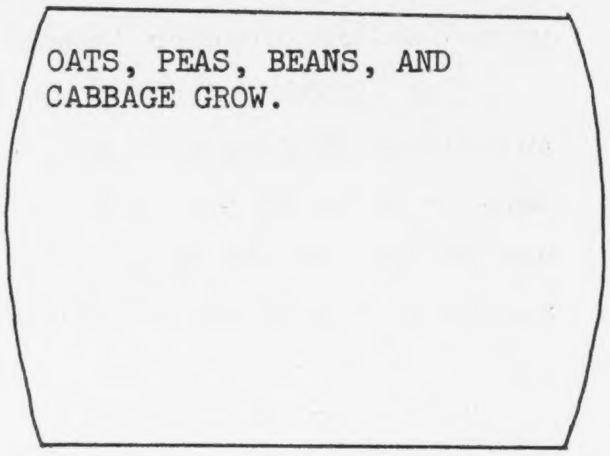


```
OATS,            PEAS,
BEANS, AND CABBAGE GROW
```

correctible by:

Statements - *WRITE OATS,;*
WRITE (NF) PEAS, BEANS, AND CABBAGE GROW.:

Result -



```
OATS, PEAS, BEANS, AND
CABBAGE GROW.
```

If WRITE (ND) text exceeds the remaining capacity of the screen, as much as can be displayed will be, and the remainder will be saved for display on the next screen.

At this point it is suggested that the reader turn to the exercises in Part VI (page 14).

ANSWER or A When ANSWER or A is encountered during execution, the arrow or carat inviting input is added to the text of the last WRITE statement, and the computer goes into "wait state." (Actually it is the ANSWER opcode which forces the preceding WRITE operand out of the display buffer and onto the screen, but this is a technical point that need not concern us at the moment.)

While the computer is in the wait state, all processing is suspended, and in fact nothing the student types on the screen will be considered by the system until he presses the "send" button. When he does, the program causes the computer to read off the characters he has typed and to store them one by one in a special area of computer memory which the DISCUS system sets aside as an "answer field." If he has typed more than 250 characters before pressing "send," all the excess is lost.

The ANSWER statement may be labelled in order to provide an address to jump to during execution, but the label is never used to refer to the contents of the ANSWER field. These are called out and displayed, or transferred to other locations simply by referring to "ANSWER".

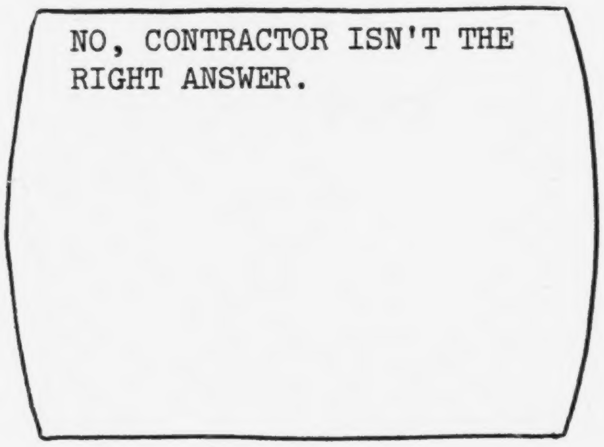
Example

Statements - *WRITE A SINGLE QUOTE ('') IS CALLED A(N)*
___________ *WHEN IT IS USED IN A CONTRACTION.;*
ANSWER;

Suppose any incorrect answer is to be quoted back to the user, and in a particular case he has typed the word "contractor." (next page)

<u>Statement</u> - *WRITE NO, "ANSWER" ISN''T THE RIGHT ANSWER.;*

<u>Result</u> -



```
NO, CONTRACTOR ISN'T THE
RIGHT ANSWER.
```

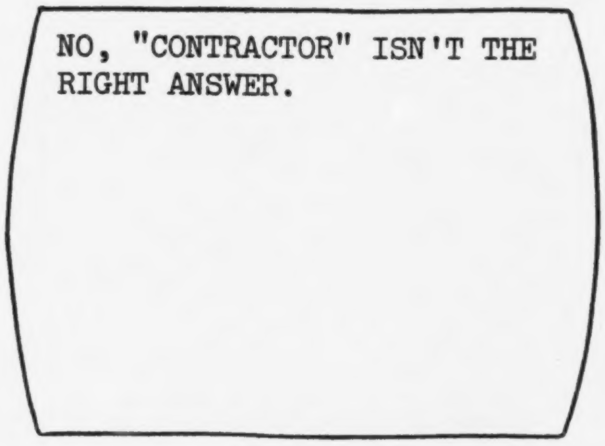
(Example continued on next page)

or better yet

Statement -

```
WRITE NO, '"'"ANSWER"'"' ISN''T THE RIGHT
ANSWER.;
```

Result -



```
NO, "CONTRACTOR" ISN'T THE
RIGHT ANSWER.
```

"ANSWER" causes only the current contents of the answer field to be displayed. In order to store an answer for future reference, it must be transferred to a labelled variable by means of a SET statement (page 57 ff.).

Note that the ANSWER statement itself, in the example on the preceding page, needs no operand. This is usually the way it is encoded, but when the programmer wishes to supply elements of the answer in order to influence the student in some way, he can do so by inserting them as operand in the ANSWER statement, thus:

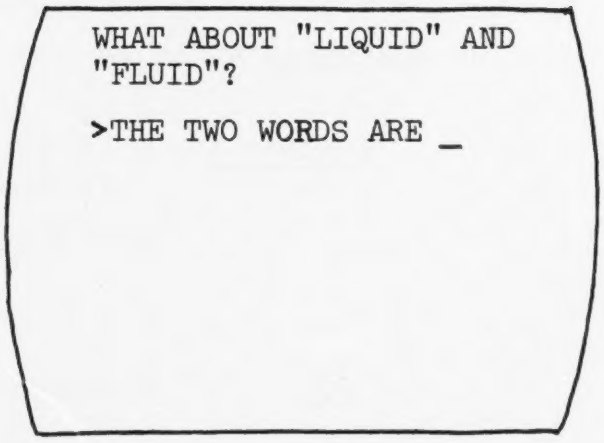
Statements -

```
WRITE WHAT ABOUT '"LIQUID" AND "FLUID"'?/;
ANSWER THE TWO WORDS ARE ;

                ------

WRITE DO YOU REALLY THINK THAT "ANSWER"?;
```

(to be used if the answer is determined to be wrong.)

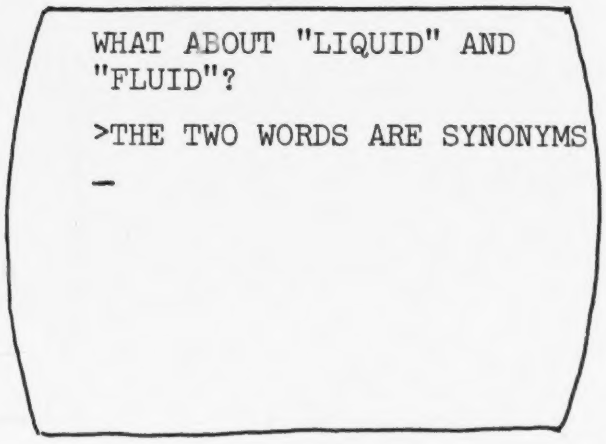
Result -

Before student starts typing:



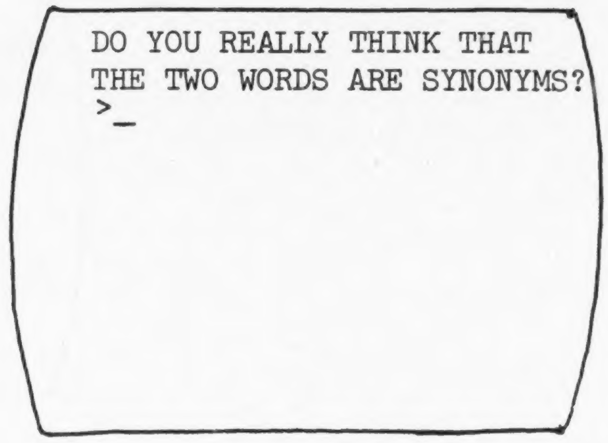
```
WHAT ABOUT "LIQUID" AND
"FLUID"?

>THE TWO WORDS ARE _
```

After typing but before "send":



```
WHAT ABOUT "LIQUID" AND
"FLUID"?

>THE TWO WORDS ARE SYNONYMS
_
```

After "send":



```
DO YOU REALLY THINK THAT
THE TWO WORDS ARE SYNONYMS?
>_
```

The foregoing device is useful in two different ways. (1) It permits giving hints outside of the basic instructional block, and these hints perform duty as reinforcers:

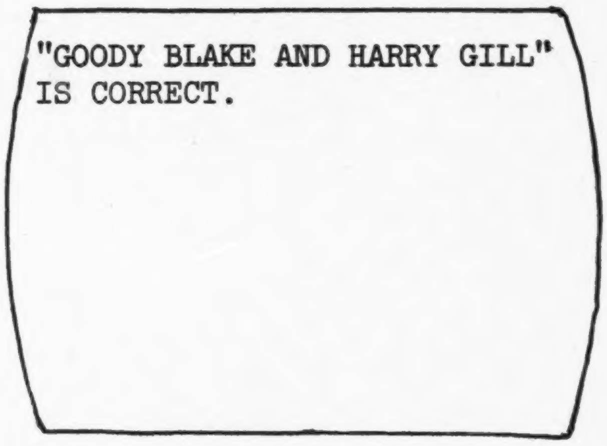
Statement - W *ONE OF THE MOST DREADFUL POEMS OF THE*
NINETEENTH CENTURY WAS WORDSWORTH''S;

A *'"'GOODY BLAKE AND;*

ⓐ W *"ANSWER"'"'IS CORRECT.:*
(to be used if student answered "Harry Gill.")

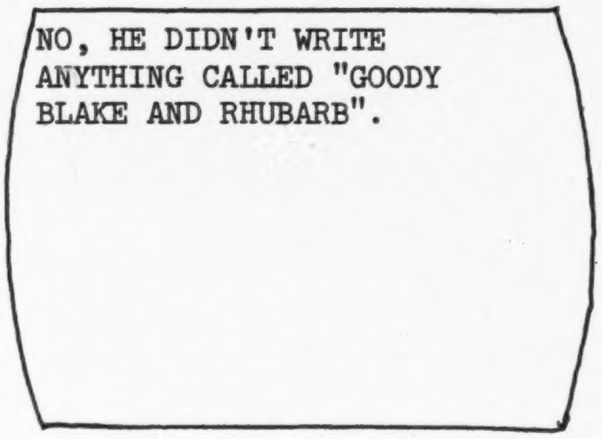
and ⓑ W *NO, HE DIDN''T WRITE ANYTHING CALLED "ANSWER"'"'.;*
(to be used if he answered something else.)

Results - ⓐ



"GOODY BLAKE AND HARRY GILL"
IS CORRECT.

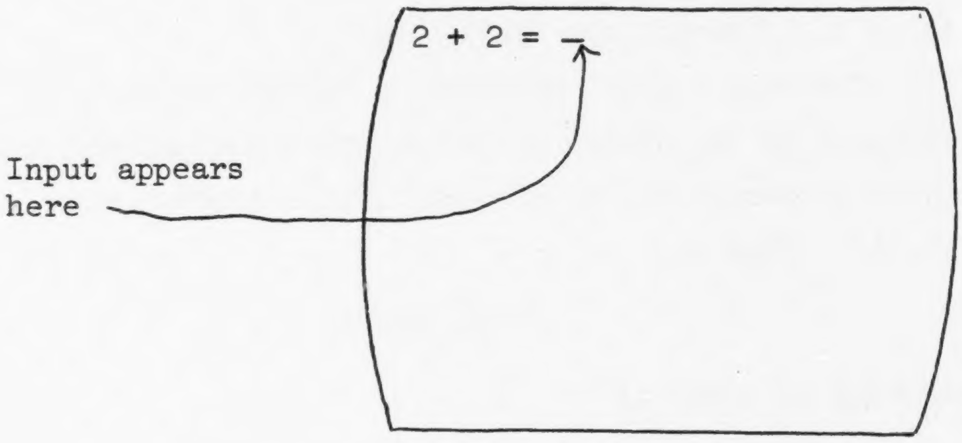
ⓑ



NO, HE DIDN'T WRITE
ANYTHING CALLED "GOODY
BLAKE AND RHUBARB".

(2) It permits pre-structuring an answer in such a way that it can be quoted later with fair assurance of a grammatical fit .

ANSWER(NF) or A(NF)   A(NF) operates in exactly the same way as the basic ANSWER opcode, except that the answer is expected at the end of the last line of displayed WRITE characters, instead of at the beginning of the next line below and no carat appears:



This permits more natural-looking displays where formulae or sentence-completion questions are involved. It does not, however, offer a means of positioning the carat within WRITE text, as might be desired in order to give realism to some types of fill-the-blank questions. A literal carat may be written in a WRITE statement but it will have no special significance to the succeeding ANSWER statement.

SCAN or SC The SCAN opcode invokes an operation which is fundamental to any "character-string match" language, i.e., one which can detect certain prescribed elements in a string of input. These elements may be single characters, whole words, phrases, sentences, punctuation marks, numerals, etc. A measure of such a language is its ability to recognize non-consecutive elements, particularly if they are in some order other than that in which they are listed in the SCAN operand; to re-examine input as many times as necessary in order to establish a certain profile; and to deal with negations. A CAI language which cannot recognize responses outside of a limited format (such as a single button-push) is of a different type altogether.

The basic scan statement in DISCUS consists of the SCAN opcode followed by an operand in which the coder enters those elements whose presence in, or absence from, student input is to be established. Thus

SC *WHITE*;

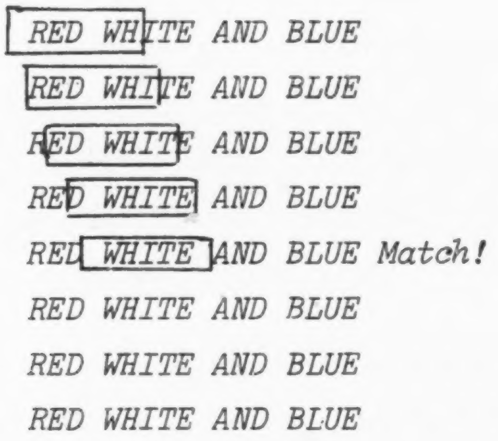
working on input of

RED WHITE AND BLUE

will be satisfied, and a system condition code will be set to indicate the fact. The information is typically used to dictate what will happen next in the execution of the program.

One way of visualizing the scan operation is to think of the operand as occupying a moving window, one which opens onto student text. It sweeps from left to right across the input string until it either encounters an uninsulated semicolon (signalling the end of the statement) or "sees" a combination of characters which match it exactly.

For example, the operand WHITE would be scanned-for sequentially as follows:



Actually what happens in DISCUS is that student input, preceded by a notation indicating its length, is stored in a field called "ANSWER" when the "interrupt" or "send page" key is pressed. The message is placed on a rack, as it were, where it can be examined in detail by the program. Only the input is thus transferred. Neither the ANSWER opcode nor the end-of-statement semicolon is moved to the ANSWER field.

The operand of a SCAN statement specifies the items to be looked for in the answer, and the order in which they will be checked. It is important that the DISCUS user understand how this takes place. What really happens when a SCAN statement is executed?

Before launching into the discussion, it will be well to consider the definitions of "word" and "literal" as given in the Glossary:

WORD	A WORD is a string of characters that does not include imbedded blanks, special characters, or symbols, and is surrounded by blanks, either explicit or implicit. WORDS used in SCAN statements constitute elements against which a user's response may be compared.
LITERAL	A LITERAL is a string of characters, punctuation marks, symbols, numerals, or explicit blanks, not used in a special code sense. In order to be treated as a LITERAL, such a string must be surrounded by single quotation marks. (For use of these marks themselves as literals, see example, page 26.) Inclusion of a character in a LITERAL suppresses any special characteristic it may normally possess in the DISCUS system.

A word is commonly recognized as a word, in written or printed communications, if it is preceded by one or more blanks and followed by one or more blanks or punctuation marks. Because this convention is almost universally accepted in the Western world, we are able to identify individual elements as elements quite rapidly, prior to interpreting them. Conceivably we could dispense with this service. For example, one is able to extract meaning from

BREAKGLASSINCASEOFFIRE

thanks to some impressive computations of which the human brain is capable. But it is much easier and faster to read (i.e., scan identify, recognize) the message given in conventional form:

BREAK GLASS IN CASE OF FIRE

Blanks really represent non-content-bearing breaks in the information stream, and although such breaks can be almost as useful to a computer as they are to humans, they are useful to it in a different way. A computer deals with blanks as definite entities, not as just vague nothings on either side of something meaningful. Perhaps the best way of defining "WORD" acceptably for both computers and humans is to say that it is a blank-less element between two blanks.

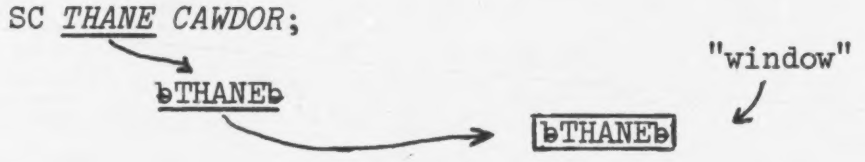
SCANNING FOR WORDS

The DISCUS system carries out the following steps in scanning for a word specified in a SCAN operand:

Step 1. The first word in the SCAN operand is identified.

Example: In SCAN *THANE CAWDOR*; "THANE" is considered to be the first word, because it is the first element not containing a blank and not enclosed in single quotes.

Step 2. The first word is moved to a location we refer to as "the window." Enroute, it is furnished with a beginning blank and an ending blank.



Step 3. The contents of the window are compared with the contents of the answer field, starting with the first string of equal length at the extreme left end of the ANSWER field, and moving one position at a time toward the right.

Suppose the ANSWER field contains

ƀTHE THANE OF CAWDORƀ

and a length notation off to one side.

(The beginning and ending blanks in the ANSWER field were added at the time student input was transferred from the terminal.)

The scan begins. In the fifth position of the "window", the contents of the window match exactly that which it sees in the ANSWER field.

ƀTHE THANE OF CAWDORƀ

ƀTHANEƀ →

If the ANSWER field had contained, instead, ƀTHE THAIN OF CAWDORƀ, the comparison would have proceeded all the way to the end of the ANSWER field, and a failure would have resulted for the SCAN operand in question.

Since we found a match on the first word in the example, we proceed with the next step, which is the same as Step 1 above, using the second word in the SCAN operand. Again, the word is furnished with a blank fore and aft, and a notation is made that we are in fact dealing with a word. However, this time the comparison begins not at the beginning of the ANSWER field, but at the ending blank of the previous successful comparison:

ƀTHE THANE OF CAWDORƀ

A (match) *ƀTHANEƀ*

B (starting position) *ƀCAWDORƀ*

C (match) *ƀCAWDORƀ*

Note that the ending blank in A overlaps the beginning blank in B. This overlap can occur only if the string being compared is a word. If the ANSWER field had contained simply ƀTHANE CAWDORƀ both the first comparison and the second comparison would have succeeded, in their first positions, even though both seem to take advantage of the same blank in the ANSWER field.

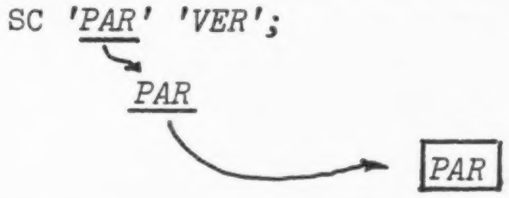
SCANNING FOR LITERALS

The DISCUS system carries out the following steps in scanning for literals specified in a scan operand:

Step 1. The first literal in the SCAN operand is identified.

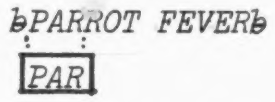
Example. In SCAN *'PAR' 'VER'*;, 'PAR' is considered to be the first literal, because it is the first element enclosed in single quotes.

Step 2. The material enclosed between the single quotes (but not the quotes themselves) is moved to the window. It is NOT furnished with beginning and ending blanks.



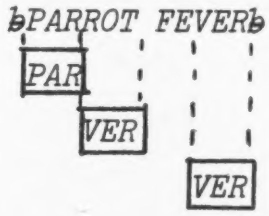
Step 3. The contents of the window are compared with the contents of the ANSWER field.

(Assume ANSWER field contains)



Match is obtained in the second position from the left.

Step 4. Repeats Step 1 (et seq.) except that the comparison begins, with the second literal, at the precise boundary of the first.



It is immaterial, in scanning for literals, whether the matching characters occur in the ANSWER field as word beginnings, word middles, or word ends. By the same token, literals may overlap more than one word in the ANSWER field; i.e., may include blanks, specified punctuation marks, etc., Thus

SC *'GT. BRIT.--HISTORY--19TH CENTURY'*;

will match the following ANSWER field:

I THINK IT SHOULD BE GT. BRIT.--HISTORY--19TH CENTURY.

It will not match a variant such as

I THINK IT SHOULD BE GT. BRIT--HISTORY--19TH CENTURY.

Such a SCAN statement is of course very rigid, whereas

```
SC 'PAR' 'VER';
```

is quite the opposite, matching such things as *"PARROT FEVER"*, *"PARLEY FOREVER"*, *"SPARE EVERY TREE"*, *"I'VE PARTED FROM VERONICA,"* etc.

We can conclude that literals do not, in themselves, make the SCAN operation any more rigid than do words, provided they are of limited length and provided they avoid specifying punctuation marks and internal blanks.

SCANNING FOR WORDS AND LITERALS INTERMIXED

Although word elements and literal elements in a SCAN operand are processed somewhat differently, there is nothing to prevent both types being used together in any combination that best suits the programmer's objective. For example, suppose he wants to find out if student input contains the words

BON HOMME RICHARD

or

BONHOMME RICHARD (either version being acceptable)

In such a case `SC 'BON' 'HOMME' RICHARD ;` would turn the trick. Or suppose he wanted to detect the name

DIONYSUS OF HALICARNASSUS

without being fussy about the spelling. He could specify

```
SC ' DIO' 'US ' OF  ' HAL' 'SUS ':
```

Note that by including blanks in the literals, the programmer effectively specifies their position as word-beginnings or word-ends. He could have specified middles:

```
SC 'ONY' OF 'CAR';
```

but with some loss of precision, in this particular case.

In both of the foregoing examples, the SCAN elements are processed in the order in which they appear, regardless of whether they are words or literals.

PUNCTUATION MARKS AND SPECIAL CHARACTERS

Before student input is placed in the ANSWER field, the punctuation marks ; : ( ) ? ! , . " are replaced with blanks.

I QUIT!!!!!	becomes	*I QUIT*
1, 2, 3	becomes	*1 2 3*
DON'T	remains	*DON'T*

If a particular punctuation mark or special character has, in the coding, been specified as a SCAN literal, the program will go back and restore it to the ANSWER field, if in fact it was typed by the student.

It should be emphasized that none of the conventions for use of single quotes, double quotes, colons, slashes, or semicolons which apply to encoded SCAN or WRITE statements affect the student's use of these marks in his input. For example, he doesn't need to type DON''T in order to have it recognized as DON'T.

ADDITIONAL EXAMPLES

Let us see how a scan operation involving more complicated strings would work. Suppose we want to scan for Don't go; stay. We would encode this in the SCAN statement as

SC *DON''T GO ';' STAY;*

Note that we have suppressed the special nature of the single quote by doubling it, and of the first semicolon by insulating it with single quotes (exactly in the same way as they are treated in WRITE operands when they are to be displayed literally (see pages 27-28)). The last character, i.e., the semicolon, remains active in its usual role as an end-of-statement delimiter.

The above scan operand will match student input only if the operand contains

DON'T
GO
;
and *STAY* in that order.

The input may be festooned with all kinds of other words and marks without spoiling the match:

HALT---DON'T GO AWAY MAD; PLEASE
STAY, WON'T YOU???

Obviously the more we demand in a SCAN operand, the less chance it has of succeeding across a given set of answers, all expressing the same idea, but expressing it somewhat differently. So we try to remove as many constraints as possible, in order to improve the likelihood of matching answers which contain the essential elements.

SUBOPERANDS

Often the coder will want to provide for a number of acceptable alternatives conveying similar meanings, any one of which will attain MATCH. In other words, he would like to perform a basic "OR" operation, as if he were testing the answer against a whole series of scans, each of which started afresh at the beginning of the ANSWER field. He cannot do this by stacking SCAN statements.

SC *6*

followed immediately by

SC *SIX*

will not lead to a match of 6 in the student input, because the second SCAN statement resets whatever condition code was set by the previous SCAN statement.

In order to describe how alternatives can be combined economically in a single SCAN statement, we need to deal with subordinate parts of SCAN operands as modular entities. It is convenient to call these "suboperands" and define them as: ANY STRING, WITHIN AN OPERAND, THAT IS SEPARATED FROM OTHER PART(S) OF THE OPERAND BY A COMMA OR AN "OR"-BAR (BOTH MEANING "OR"), OR BY AN AMPERSAND (MEANING "AND").

Example:

$$\text{SC}\quad \overbrace{\underbrace{\textit{CORN,}}_{\text{suboperand}}\ \underbrace{\textit{MAIZE}}_{\text{suboperand}}}^{\text{operand}}\quad ;$$

Operation: The program scans the ANSWER field first for CORN; if MATCH is obtained, it sets the condition code and jumps to the next statement, disregarding the second suboperand. If no match is obtained on CORN, it re-scans the ANSWER field for MAIZE. If match is obtained, it sets the condition code accordingly. Execution then passes to the next statement. If neither is found, the condition code is set to FAIL and execution continues.

Example: (4 suboperands)

SC *OATS, PEAS | BEANS | BLACK EYED PEAS;*

("or"-bars)

Operation: The program scans for each suboperand in sequence, desisting only when success is attained or the end-of-statement delimiter is reached.

Two or more suboperands separated by (an) ampersand(s) will both (all) be scanned before success is assured, match condition code set,* and scan terminated. The order in which the suboperand elements appear in the answer is immaterial:

Example:

$$\text{SC}\quad \overbrace{\underbrace{\textit{SLEET}}_{\text{suboperand}}\ \&\ \underbrace{\textit{SNOW}}_{\text{suboperand}}}^{\text{operand}}$$

Operation: The program scans the answer field twice, first for SLEET, then for SNOW. Match condition code is not set unless both are found.

This permits scanning for full permutations of lists where order is unimportant.

Example: SC *FIRE & WATER & EARTH & AIR;*

Operation: Four separate scans are performed. Failure on any one of them terminates the operation, while success is not determined until all four have been scanned. If every possible combination had to be set up as a separate operand or suboperand (SC FIRE WATER EARTH AIR, WATER EARTH AIR FIRE, etc._ _ _ .;) twenty-four of them would be required, and the whole series would have to be performed to establish the absence of one of the required terms.

---

* Condition codes are explained in THE DECISION PROCESS (page 67).

Any combination is permissable:

FIRE & COLD WATER, EARTH & AIR, FISH CHIPS|
CAKES ALE: (six suboperands)

Operation:

1. Scan for FIRE. If unsuccessful, go to next suboperand preceded by (|) or (,).
   If successful,.......
2. Scan for COLD and WATER - in that order.* If successful, set match condition code, jump to next statement (beyond the semicolon). If unsuccessful, go to the next suboperand preceded by (|) or (,).......
3. Scan for EARTH. If unsuccessful, go to next suboperand preceded by (|) or (,).......
   If successful.......
4. Scan for AIR. If successful, set match condition code, skip to next statement. If unsuccessful, go to next suboperand preceded by (|) or (,)....
5. Scan for FISH and CHIPS, in that order, with any number of characters and/or blanks intervening. If successful, set match condition code, jump to next statement. If unsuccessful, go to next suboperand preceded by (|) or (,).......
6. Scan for CAKES and ALE, in that order with any number of characters and/or blanks intervening. If unsuccessful, go to next statement.

---

*Additional intervening blanks in the answer will not cause failure of this scan. Neither will intervening characters, as long long as they don't adjoin either of the specified words. For example,

COLD WATER COLD WATER COLD BLUE WATER	would succeed
WATER COLD	would not succeed

From the above examples we can deduce a rule: namely, that

After scanning any suboperand ending with a

comma, or |	success terminates the operation failure goes to the next suboperand
ampersand	success goes to the next suboperand failure goes to the next suboperand which is preceded by a comma or "or"-bar.
semicolon	terminates the operation

SPOILERS (¬)

In mechanics, a "spoiler" is an attachment that reduces or neutralizes the effect of a device, such as deflection vanes used to reduce the lift of an airplane wing, the tinting blended into windshield glass, the mute on a trumpet. In DISCUS, we use the term in a more absolute sense, to apply to the "not" sign in a scan operand.

SC *¬CATS;*

will match any answer field that does not contain the word CATS.

SC *¬CATS, DOGS;*

will work as follows:

Answer field	Result
DOGS ARE FRIENDLY CRITTERS	Succeeds on not CATS
I LIKE CATS AND DOGS	Succeeds on DOGS
I LIKE DOGS AND CATS	Succeeds on DOGS
CATS ARE EGOCENTRIC	Fails on not CATS and the absence of DOGS.

If one changes the order of the suboperands to

SC *DOGS, ¬CATS*

the same results are obtained, but according to a different progression:

DOGS ARE FRIENDLY CRITTERS	Succeeds on DOGS
I LIKE CATS AND DOGS	Succeeds on DOGS
I LIKE DOGS AND CATS	Succeeds on DOGS
CATS ARE EGOCENTRIC	Fails on the absence of DOGS, and then again on not CATS.

Clearly the intent of a spoiler and the words to which it applies is best served by placing them in a suboperand ahead of a desired element.

An important distinction must be made between the effect of a spoiler in multi-word suboperand, and in suboperands which are separated by ampersands. In the former, the spoiler - regardless of its location - applies to each word in the suboperand. In the latter case, only to the word or words in the suboperand which contains it.

With either ....
SC ¬ *CATS DOGS*;
SC *CATS*¬ *DOGS*;
(These two statements are equivalent.)

Answer field	Result
DOGS ARE FRIENDLY CRITTERS	Failure
I LIKE CATS AND DOGS	Failure
I LIKE DOGS AND CATS	Failure
CATS ARE EGOCENTRIC	Failure
I LIKE HORSES	Success

SC ¬ *CATS & DOGS*

Answer field	Result
DOGS ARE FRIENDLY CRITTERS	Success
I LIKE CATS AND DOGS	Failure
I LIKE DOGS AND CATS	Failure
CATS ARE EGOCENTRIC	Failure
I LIKE HORSES	Failure

(Reversing the order has no effect, because both elements are tested before scanning terminates.)

Thus if we wanted to match

I prefer Hindemith to Chaminade but not
I prefer Chaminade to Hindemith

we would encode the scan statement either as

SC *HINDEMITH CHAMINADE* & ¬ *CHAMINADE HINDEMITH*;

or as

SC *HINDEMITH CHAMINADE & CHAMINADE ¬ HINDEMITH*;

But if we didn't care which he preferred as long as he specified both but didn't mention the Scarf Dance, we could encode it

SC *¬SCARF & HINDEMITH & CHAMINADE*

The "not" facility of DISCUS is a very powerful tool, permitting one to pack into one SCAN statement a number of parameters that ordinarily would require a whole series of statements, each needed to detect a specific undesirable element before getting down to the desired element and its alternatives.

The "¬" also can be used to detect simple negation, in order to reduce one particular risk of misinterpreting input:

Question	W	*WHO IS PRESIDENT OF THE UNITED STATES*	
(if)Answer	A	*NOT NIXON*;	
Simple scan	SC	*NIXON*;	(Success)

The program reaction in such a case would be exactly opposite to the one intended. Scanning the same input with a ¬ spoiler:

SC *¬NOT & NIXON*; (Failure)

To cover a greater variety of possibilities, the scan would probably be written

SC *¬ NOT & NIXON ¬ AIN''T & NIXON ¬ DON'T & NIXON ¬ NO & NIXON*;

[(A Boolean expression such as

(¬ NOT, ¬ AIN''T, ¬ DON''T, ¬ NO) & NIXON

is not within the SCAN capability of DISCUS as currently implemented.]

The same device could be used to handle double or even triple negatives in cases wherein these occurred as separate words, but simple reversals of meaning do not occur consistently enough in English to make this a very productive gambit. A really sophisticated "not facility" should be able to unravel negative affixes as well as stand-alone forms of "not", so that the true polarity of meaning in a sentence could be established.

EXPANDING CONTENTS OF VARIABLES INTO OTHER STRINGS

We have seen how DISCUS scans for elements specified in the SCAN operand. Let us now consider scansion for elements which are only referred-to in the operand.

The contents of a variable statement may be referred to by another statement only through its address, i.e., its label. Thus

WRITE   *OH, THAT THIS TOO, TOO SOLID "ICECUBE" WOULD MELT*;

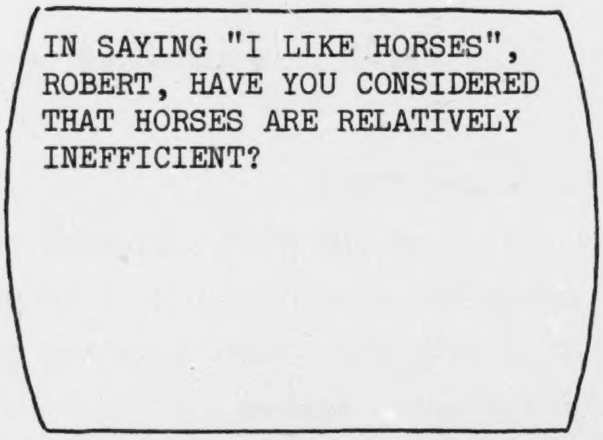
writes the following on the screen

OH THAT THIS TOO TOO SOLID FLESH WOULD MELT

only if somewhere in the program there exists a variable whose label is ICECUBE: and that contains the single word FLESH. If ICECUBE contained a laundry list, the whole thing would be displayed between SOLID and WOULD.

With a WRITE statement the contents of a variety of labelled data may be thus expanded into the operand. Since the ANSWER field is a variable, its contents may also be called forth at any time, but this should be done by referring to "ANSWER" rather than to a label which may have been attached to the ANSWER statement itself in a particular frame.

Thus WRITE *IN SAYING'"' "ANSWER" '"', "NAME", HAVE YOU CONSIDERED THAT "HORSES"?*; would be displayed as



IN SAYING "I LIKE HORSES",
ROBERT, HAVE YOU CONSIDERED
THAT HORSES ARE RELATIVELY
INEFFICIENT?

provided   *I LIKE HORSES* is contained in the answer field,
ROBERT is contained in a variable labelled NAME,
and   *HORSES ARE RELATIVELY INEFFICIENT* is contained in a variable labelled HORSES.

Straight text may be set into a variable and left unchanged; it is still possible to scan for it - remembering, of course, that the chances of obtaining a match on exact text are fairly slim.

At any given moment a variable may contain either a number (expressed as an integer) or a string of characters. The former might be scanned-for in the answer field if we wanted to know if the student had the right answer to a simple arithmetic problem he himself had constructed. The latter might be scanned for as a standard element expected to occur in a great many responses, or to check a unique item supplied earlier by the student himself.

The entire length of a character variable expanded into a SCAN operand is treated by DISCUS as a literal. This means that imbedded commas, or-bars, and ampersands do not serve as logical operators in the SCAN statement.

In a variable character string consisting of A,B,C the commas will be passive if the string is used in a SCAN operand.

So in SC *Q, "CC",X*, where CC is the address of the above string the operand becomes

Q, A,B,C, X
↗ ↖↗ ↖
active passive active

and match will be obtained only if

Q or
A,B,C (including the commas) or
C

is present in the answer field.

Further discussion of the SCAN statement must be deferred until Part III, after the remaining DISCUS OPCODES have been covered, in order to give the reader a better idea of how they behave as part of the total system.

VARIABLES

There are no standard constants imbedded in DISCUS. For example "pi" does not automatically conjure up 3.1416. If needed in a particular program it would first have to be submitted as part of the source. For convenience we would establish it as a "variable," even though we probably wouldn't plan to make any changes in it.

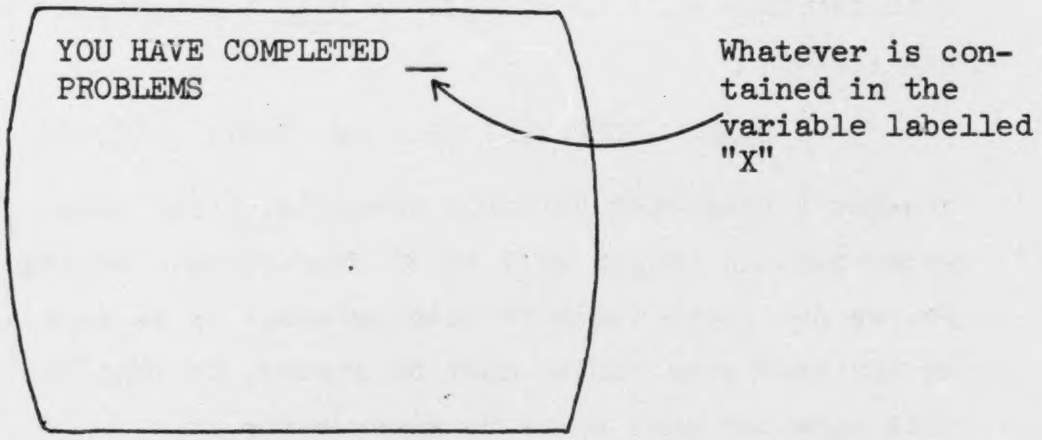
A variable can be thought of as a box with a label on it, into which we can stuff things, later examining or copying their contents, adding to them, or emptying them out entirely. We can be quite sure of the exact contents of the box, provided we put them there in the original coding and provided we allow no change to occur during execution.

More frequently, however, we will want them to play a more dynamic role in on-line operations. At any given moment during execution, only the computer will know exactly what is in a variable used in this way. Neither our source program nor the object code listing will show it.

The only way to gain access to a variable, specifically to its <u>contents</u>, is by referring to its label. In making such references, the label must always be surrounded by double quotes. Thus

```
WRITE    YOU HAVE COMPLETED "X" PROBLEMS;
```

will cause display of



Other uses:

Statement - A "X";

Result - Causes the contents of X to be displayed following the carat. After the student types his answer and pushes the "send" button, the contents of "X" will be entered in the ANSWER field along with his input.

Statement - SET "X" = "Y";

Result - Causes the contents of X to be made identical with the current contents of Y.

Statement - TEST "X" = 2;

Result - Causes the contents of X to be tested to see if they are equal to 2.

(SET and TEST statements are discussed in detail in the following pages.)

DEFINE(A) or D(A)
DEFINE(C) or D(C)

To create a DISCUS variable, it must first be defined, either as an arithmetic variable or as a character variable. And a label must be assigned to it at that time.

ADDO: DEFINE(A); or ADDO: D(A);

creates a numerical variable with a capacity from zero to over ± two billion (to be exact, an IBM/360 full word integer). This variable will be accessible only through its unique label - ADDO.

CHAT: DEFINE(C) 30; or CHAT: D(C) 30;

creates a character variable whose label (or name) is CHAT and whose maximum length will be 30 characters. We can specify as few as one (zero would be meaningless,) or as many as 255, but in any case some number must be stated, so that the compiler will know how much space to reserve for it.

Neither of these two forms of DEFINE initialize the name variables. Nothing is put into either box. To do that a SET statement is used.

SET or S SET uses the basic programming device of altering the contents of a variable of a field by restating its contents with some change, if a change is desired. Using the examples on the preceding page, if it were desired to start ADDO off containing 10, we would simply follow the DEFINE statement with

SET *"ADDO" = 10*;   or   S *"ADDO" = 10*;

(the SET statement does not have to follow directly - there could be a long list of DEFINEs followed by a long list of SETs.) After thus "initializing" ADDO, we can change its contents directly by restating them to be another number or to be the current quantity plus or minus something else, or divided by something, or multiplied by something, etc., but in any case the quantity to the right of the equal-sign - which is never omitted - is always the new value of that variable.

If we desired that CHAT begin its career with the letter Z as its contents, the statement

SET *"CHAT" = 'Z'*;   or   S *"CHAT" = 'Z'*;

would take care of the matter. Subsequent changes are effected along the same lines as with arithmetic variables; that is, the contents are restated either completely or by adding to the existing contents, as expressed to the right of the omnipresent equal sign. To eradicate the contents of either type of variable, one encodes nothing at all to the right of the equal-sign (not zero, which is something different from nothing). Thus the two variables above could be collapsed by

SET *"ADDO" = ;*   and   SET *"CHAT" = ;*

(although they still exist - with null contents).
Note that the address label of a variable referred to in a SET statement is always enclosed in double quotes. This is consistent with the general rule that in order to refer to the contents of a variable through that variable's label, the label must be enclosed in double quotes.

SET establishes, in a variable, a specified combination of characters or numbers or other variables. The operation is always performed on the variable specified to the left of the equal-sign.

Statements -
```
CUPCAKE:  D(A);
MUFFIN:   D(A);

          SET "CUPCAKE" = 3;
          SET "MUFFIN"  = 4;
          SET "CUPCAKE" = "CUPCAKE" + "MUFFIN";
```

Result - "CUPCAKE" now contains the number 7.
"MUFFIN" still contains the number 4.

Statement -
```
SET "CUPCAKE" = "MUFFIN";
```

Result - Both "CUPCAKE" and "MUFFIN" contain 4.

Statement -
```
SET "MUFFIN" = "MUFFIN" + 20 - "CUPCAKE";
```

Result - "MUFFIN" contains 21. "CUPCAKE" still contains 4.

The arithmetical operators are

+	add
-	subtract
*	multiply
/	divide

The arithmetic operations are performed sequentially, from left to right. The results of each separate operation is progressively rounded off to the lesser integer before going on to the next operation (i.e., truncated).

Statement -
```
SET "MUFFIN" = "CUPCAKE" * 7 + 1 / 3;
```

Result - "MUFFIN" contains 9, not 9.66666 or 9 2/3.
(4 x 7 = 28; 28 + 1 = 29; 29 ÷ 3 = 9 2/3; lesser integer = 9)

The nature of the variable to be set, i.e., the one to the left of the equal-sign in the SET operand, determines the nature of the operation to be performed. If it was originally defined with a D(A), the operation will be arithmetical:

Statement - SET *"CUPCAKE" = 365 + 144;*

Result - "CUPCAKE" contains 509, not 365144.

If through error or otherwise there is an attempt to add non-numeric characters to an arithmetic variable, they will cheerfully be disregarded.

Statement - SET *"CUPCAKE" = 365 + 'DAYS';*

Result - "CUPCAKE" contains 365 (not 365DAYS or 3650.)

In a character variable - one defined by a D(C) statement - no arithmetic operation takes place, regardless of whether numbers or characters are used.

Example

Statements - PIE: D(C) 30;

```
                  S     "PIE" = '4';
                  S     "PIE" = "PIE" ' AND' ' 20' 'BLACKBIRDS';
```

Result - PIE contains

4 AND 20 BLACKBIRDS (not 24 AND BLACKBIRDS)

Note that character strings are entered in SET operands as literals, i.e., surrounded by single quotes, if they are entered directly, as in the example above. If they are entered indirectly i.e., by reference to the contents of another labelled variable, the label specified in the operand is enclosed in double quotes.*

If a blank is needed in order to separate two literals, it must be specified. Hence ' AND' and ' 20' and ' BLACKBIRDS' above, not 'AND' and '20' and 'BLACKBIRDS'.

Non-literal blanks mean nothing at all in the operand:

```
S  "PIE" = "PIE" ' ' 'AND' ' ' 'BLACKBIRDS';
```

and

```
S  "PIE" = "PIE"      ' '   'AND'   ' '   'BLACKBIRDS';
```

would have identical results.

If the operand were written

```
S  "PIE" = "PIE"     'AND'    '20'   'BLACKBIRDS';
```

the resultant content of PIE would be

4AND20BLACKBIRDS.

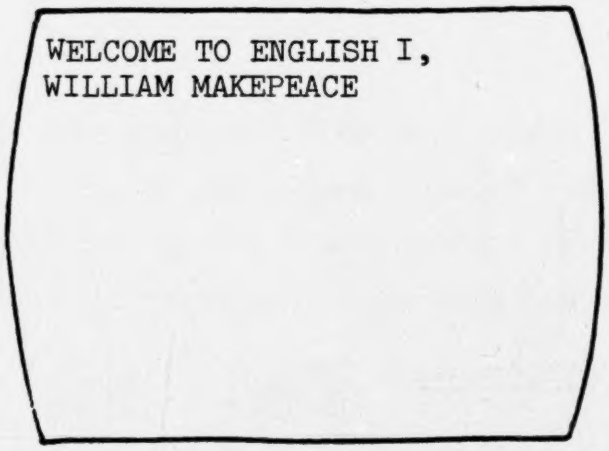
The SET command provides a means of saving student answers for later user, but the maneuver must be performed before the next ANSWER statement is encountered, since every new answer annihilates whatever preceded it in the ANSWER field.

---

*This follows the general rule that the contents of a labelled statement are always referred to by specifying that label. When referring to the contents of a variable, its name (label) must, in addition, be enclosed in double quotes.

The contents of the current ANSWER field may be added to the existing contents of a variable either ahead of or following them, or inserted between two or more specified elements. In the following example we add the student's last name to a previously-saved first name:

```
Statement  -  W  WHAT IS YOUR FIRST NAME?;
              A; (student types WILLIAM)
              S  "NAME" = "ANSWER";
              W  THANK YOU. NOW YOUR LAST NAME,* PLEASE.;
              A; (student types MAKEPEACE)
              S  "NAME" = "NAME" ' ' "ANSWER";
              W  WELCOME TO ENGLISH I, "NAME";

Result     -  (contents of current ANSWER field added on the
              right)
```



```
WELCOME TO ENGLISH I,
WILLIAM MAKEPEACE
```

In the following example we add the student's first name to a previously-saved last name:

```
Statement  -  W  WHAT IS YOUR LAST NAME?;
              A; (student types MAKEPEACE)
              S  "NAME" = "ANSWER";
              W  THANK YOU. NOW YOUR FIRST NAME, PLEASE.;
              A: (student types WILLIAM)
              S  "NAME" = "ANSWER" ' ' "NAME";
              W  WELCOME TO ENGLISH I, "NAME";

Result   - -  The same
```

If the original DEFINE statement for a character variable has reserved too little space to accommodate the material which one later tries to SET into it, the SET statement which exceeds the available space does not fail altogether, but manages to enter as many characters as possible, starting from the left end of the input string. The remainder vanish. This could be useful if the encoder wanted to throw away all but the first n characters of an answer, but more commonly the D(C) should specify ample room for everything expected to go into it.

---

* Assume "NAME" has been previously defined as a character variable.

TEST or T The TEST statement compares two variables with each other, or a variable with a literal, and sets a condition code* to positive if "successful" according to the terms of the relational operator (equals = ; greater than > ; less than < ; not ¬ . They may be clustered or used singley.) As with SET, the object of the comparison is the element to the left of the relational operator.

```
                    TEST "A" > "B";
```

will result in a positive condition if A is in fact greater than B.

The type of comparison is determined by the object variable. If it is arithmetical, the second variable is dealt with as such, if at all possible. For example, suppose the following is attempted:

```
Statements - CUP:      D(A);
             SAUCER:   D(C) 20;

                       S    "CUP" = 10;
                       S    "SAUCER" = 'BUTTERFIELD 8';

                       T    "CUP" > "SAUCER";

Result       - Positive
```

The program will compare the contents of CUP with the 8 in SAUCER, and disregard the alphabetical characters and the blank.

If the object variable is a character variable and the second variable is arithmetic, the contents of the latter will be dealt with as a character string.

---

*The use of condition codes is explained in THE DECISION PROCESS, p. 67.

Example

```
Statements  -  X:       D(A);
               Y:       D(C) 20;

                        S   "X" = 456
                        S   "Y" = '45';
                        S   "X" = "Y" '6';
                     TEST   "X" = "Y"; will succeed.
```

If both variables are character variables, then the smaller of the two is padded with blanks on the right, and a character-by-character comparison takes place.

- -

Examples

```
(Assume "Q" is a numerical variable containing 4
   "    "R"  " "    "           "          "     6
   "    "S"  " " character      "          "     'GRAVY')
```

TEST "Q" = "R"; will fail, because 4 is not equal numerically to 6.

TEST "R" > "S"; will succeed, because GRAVY contains no number greater than 6 (in fact, no number at all.)

TEST "S" ¬ > = "Q"; will succeed, because GRAVY is neither greater than nor equal to '4'.

In the following example we test "S" against another character variable "T" containing 'GRA':

TEST "S" = "T"; will fail, because 'GRAVY' is not the same as GRA.

- -

A strong similarity exists between TEST and SCAN. SCAN is really a kind of moving TEST, which sweeps from left to right across the variable being tested, i.e., the ANSWER field. The SCAN relational operator - in this analogy - is always = .

Both TEST and SCAN result in the setting of a condition code to indicate success or failure, so that a determination can be made as to what will happen next in the execution of the program.

To illustrate these operations, two examples are needed:

Ⓐ To verify the exact contents of a variable (TEST)

Ⓑ To ascertain if a character variable contains one or more specified elements. (SCAN)

Ⓐ Suppose the student is asked to furnish the last four digits of his social security number in order to use it as a code number on restart. Either a character variable or a numerical variable could be used to store these four digits, but it would be better to use a numerical variable in order to eliminate "noise". Compare:

SEC: D(A);	SOC: D(C) 20;
(Student types "OK. 5321")	(Student types "OK. 5321")
S *"SEC" = "ANSWER";*	S *"SOC" = "ANSWER";*
Result:	
SEC contains 5321	SOC contains OK. 5321
because whenever an attempt is made to put characters into an arithmetic variable, they are automatically thrown away, without affecting the numerical entry.	

When the student signs in for his next session, the system will bring his data set out of storage. The program can then ask him to confirm his identity by again giving the last four digits of his social security number. Suppose this time he types "5321".

Result:	
T *"ANSWER" = "SEC";* will succeed.	T *"ANSWER" = "SOC";* will fail.
	If SOC had been limited to four characters by
	SOC: D(C) 4:
	it would contain OK.ƀ

and still fail. One could check SOC immediately after the original input to see if it did in fact contain a four-digit number, but the routine for this is fairly tricky and (machine) time consuming.

On the other hand

it would be a good idea to check SEC immediately after the original input to see if it contained a four-digit number, as follows:

```
T "SEC" > 9999;
M;
W JUST FOUR DIGITS, PLEASE.;
J (label of ANSWER statement);
E; *
T "SEC" =< 1000;
W FOUR DIGITS, PLEASE.;**
J (label of ANSWER statement);
E;
```

Ⓑ In this example suppose we have been stuffing words from successive answers into a character variable defined as

CITIES: D(C) 250;

and we would like the program to be able to ascertain - at some time during execution - if the variable contained either MEMPHIS or CHATTANOOGA. Since TEST tests for all or nothing at all, it won't work here. But we can use SCAN for the purpose: (Assume we have another character variable called "TEMP" lying around, which we can use temporarily to store the current contents of the ANSWER field while we use the ANSWER field for SCANning something else.)

---

* Use of these opcodes is explained in THE DECISION PROCESS, page 67.

** Actually, one or more additional tests would be needed here to check for leading zeros, e.g., to validate a four-digit number such as 0043.

```
                S "TEMP" = "ANSWER";
                S "ANSWER" = "CITIES";
                SC MEMPHIS, ' CHAT' 'GA ';
                F;*
                W YOU HAVEN''T MENTIONED EITHER MEMPHIS
                  OR CHATTANOOGA.;
                S "ANSWER" = "TEMP";
                E;
```

Or, turning the procedure around in order to detect repetitions:

```
TENN:           A;
                SC MEMPHIS, ' CHAT' 'GA ';
                M;*
                S  "TEMP" = "ANSWER";
                S  "ANSWER" = "CITIES";
                SC MEMPHIS, ' CHAT' 'GA ';
                M;*
                W  YOU ALREADY MENTIONED "TEMP";
                J  TENN;
                E;*
                W  CORRECT.;
                S  "CITIES" = "CITIES" ' ' "TEMP";
                S  "ANSWER" = "TEMP";**
                E;*

                        etc.
```

There is no need to restore "ANSWER" at this point, because the next input will clear it.

---

*Use of these opcodes is explained in THE DECISION PROCESS, page 67.

**Actually, there is no need to restore "ANSWER" at this point, because the next input from the student will clear it.

THE DECISION PROCESS In DISCUS, the decision process can lead to a number of actions such as

"jumping" to another location in the program and resuming execution at that point;

displaying the contents of a variable;

changing the contents of a variable;

"using" a section of code located at some other part of the program;

displaying a string of text;

causing the DISCUS system to await student input;

invoking author mode (discussed on pages 137-140);

automatically terminating execution;

or simply passing execution to the next statement in sequence.

To illustrate, suppose SCORE represents an arithmetic variable which the course author wishes to increment every time the student answers a question correctly. Then let us imagine a successful match on the following:

```
SC BATS;
```

As a result, we want to process the following statement:

```
SET "SCORE" = "SCORE" + 1;
```

However, before the program encounters the SET statement, we will want to insert a proviso that it be executed only if a match in fact did occur; we don't want the SET statement to be executed otherwise. To provide for this selective treatment, a new DISCUS statement is used, its opcode being MATCH, or M. It needs no label or operand. It simply says to the computer, "If the current condition code - as set by the most recently executed TEST or SCAN operation - is positive, let the next operation take place. Otherwise, skip it."

So the sequence would be encoded:

```
SCAN BATS;
MATCH;
SET "SCORE" = "SCORE" + 1;
```

The same kind of arrangement can govern other actions:

```
TEST "SCORE"  8;
MATCH;
WRITE YOU ARE DOING VERY WELL.  YOUR CURRENT
     SCORE IS "SCORE".;
```

The program is not limited to doing only one thing per test or match:

```
SCAN HELEN OF TROY;
MATCH:
SET "SCORE" = "SCORE" + 1;
SET "TROJAN" = "ANSWER";
```

We can now close off this little routine and then do something about the failures. An END statement - again consisting solely of the opcode - is used:

```
END;     ( or E; )
```

This can be followed by an action which will always take place if there has been no match, in effect saying to the computer, "If the current condition code is <u>not</u> positive, let the following operation take place:"

```
FAIL;
WRITE NO. IT WAS HELEN OF TROY;
SET "SCORE" = "SCORE" - 1;
JUMP (to some other label address);
END;
```

MATCH AND FAIL COUNTERS

One can limit the number of times a given MATCH or FAIL statement will operate during processing of the section of code wherein it is located, by adding a blank, followed by a limiting numeric expression, to the MATCH or FAIL opcode. Actually, this limiting value is part of the operand of the MATCH or FAIL statement.

MATCH 1;	or M 1;	will be processed no more than once, then suppressed, along with all statements that depend on it.
FAIL 2;	or F 2;	will be processed no more than twice, then suppressed, along with all the statements that depend on it.

The way in which this feature may be exploited will be demonstrated after we have considered the <u>decision process</u>. For direct reference, see page 82 .

JUMP or J

JUMP is fairly self-evident. A JUMP statement transfers control unconditionally to the opcode associated with a label that has been specified in the operand of the JUMP statement. Thus *JUMP RUNCIBLE;* will cause processing to switch to that part of the program where a statement labelled RUNCIBLE is to be found, and to resume sequential processing at that point. Note that the destination-label is *not* in quotes. (ONLY the labels of *variables* should be in double quotes when they are referred to in the operand of a statement.)

The JUMP statement itself may, of course, be labelled, e.g., SPOON: JUMP RUNCIBLE; (If at the destination we were to find RUNCIBLE: JUMP SPOON;, an infinite loop would result. DISCUS squelches infinite loops after a large fixed number of cycles.)

JUMP is arbitrarily suppressed when it would, if executed, violate condition code levels in a block structure. This will be explained after we consider the *decision process*.

RECAPITULATION

Thus far we have discussed a number of OPCODES and the statements which they govern:

WRITE	or	W	(p. 22)
WRITE(NF)	or	W(NF)	(p. 30)
WRITE(ND)	or	W(ND)	(p. 31)
ANSWER	or	A	(p. 34)
ANSWER(NF)	or	A(NF)	(p. 39)
SCAN	or	S	(p. 40)
DEFINE(A)	or	D(A)	(p. 56)
DEFINE(C)	or	D(C)	(p. 56)
SET	or	S	(p. 57)
TEST	or	T	(p. 67)
JUMP	or	J	(above)

It has been stated that SCAN and TEST govern the decision process, and in demonstrating this we have briefly touched upon four additional OPCODES:

MATCH	or	M	(p. 68)
FAIL	or	F	(p. 68)
END	or	E	(p. 68)

The intricacies of MATCH, FAIL, and END can be properly understood only after a general discussion of the "block structure," which follows.

THE BLOCK STRUCTURE

In order to appreciate what a block structure in CAI can do for us, it is necessary to review the factors which have led to its development.

CAI programs and the "language" systems in which they are implemented can be described generally along the following lines:

Step 1	A chunk of instructional text is displayed.
Step 2	A question is displayed
Step 3	The student answers the question
Step 4	The answer is checked against pre-specified strings in order to determine a. if it is "correct" b. if it is "incorrect" c. if it cannot be recognized as either
Step 5	Program reactions unique to 4a, 4b, and 4c are arranged for each case, such as the display of text which purports to comment on the answer the display of text which suggests or states the answer sought the incrementing of a score variable jumping to the next chunk of instructional text jumping to another part of the program.

The ease with which a CAI mechanism can be described along these lines can lead to its being oversold on precisely those points where it is often weakest: perceptiveness (Step 4) and versatility (Step 5.) Efforts to make CAI programming as simple as possible can lead to routines that work in only one way, whereas the intellectual aspects of a unit of dialogue may call for elaborate options....at least if the idea of dialogue is to be sustained at all. Otherwise it is a little like trying to paint a picture without being allowed to mix the colors.

The situation is typified in connection with the program reactions listed in Step 5, above. A very simple CAI language is limited to working in one or the other of two basic condition-code settings, and is governed through a whole series of operations in a given frame by that one setting. Such frames tend to be sequence-bound, that is, reactions to matches must - both in the source coding and in the object module - precede the reactions to non-matches. There are usually ways to program around these constraints, but this sort of effort is not a happy investment.

DISCUS reduces opcodes to their simplest terms, each designed to do one thing and to do that one thing invariably - regardless of condition code setting. Then it provides two special opcodes whose sole task is to act as gates - one of them opening only if the condition code which governs it has been set to indicate a "success," the other only if the condition code which governs it has been set to indicate "failure." These are the MATCH and FAIL opcodes.*

Instead of a command meaning "WRITE IF A MATCH HAS BEEN MADE" we use the command WRITE or W, meaning "WRITE IF YOU GET A CHANCE", and ahead of it place a MATCH statement which will bar execution of the WRITE unless the condition code setting at the moment indicates "success." *

In the same way, instead of a command saying "WRITE THIS IF NO MATCH HAS BEEN MADE," we again use WRITE, but precede it with a FAIL statement to prevent execution if the condition code is set to other than "fail."

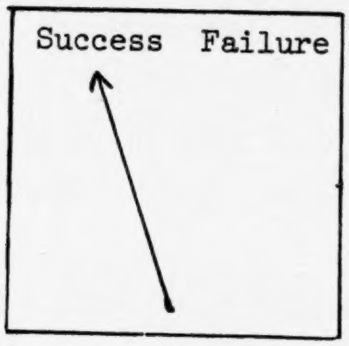
This may be done across a range of elements, each of which is to be checked. Or it may be done on a contingent basis, i.e.,

Block level of nesting	
1	Check for A
2	If A is true, check for B, otherwise pass to point Z
3	" B " " " " C, " " " " Y
4	" C " " " " D, " " " " X
5	" D " " " " E, " " " " W
6	" E " " " " F, " " " " V
7	" F " " " " G, " " " " U
8	" G " " " " H, " " " " T
9	" H " " " " I, " " " " S
10	" I " " " " J, " " " " R
11	" J " " " " K, " " " " Q
12	" K " " " " L, " " " " P
13	" L " " " " M, " " " " O.
(to maximum dept of 255)	

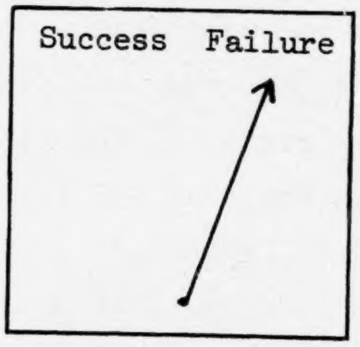
---

*Professional programmers may object to the terminology, but for purposes of this Manual we equate the words "success" = MATCH = "positive" ; "failure" = FAIL = "negative".

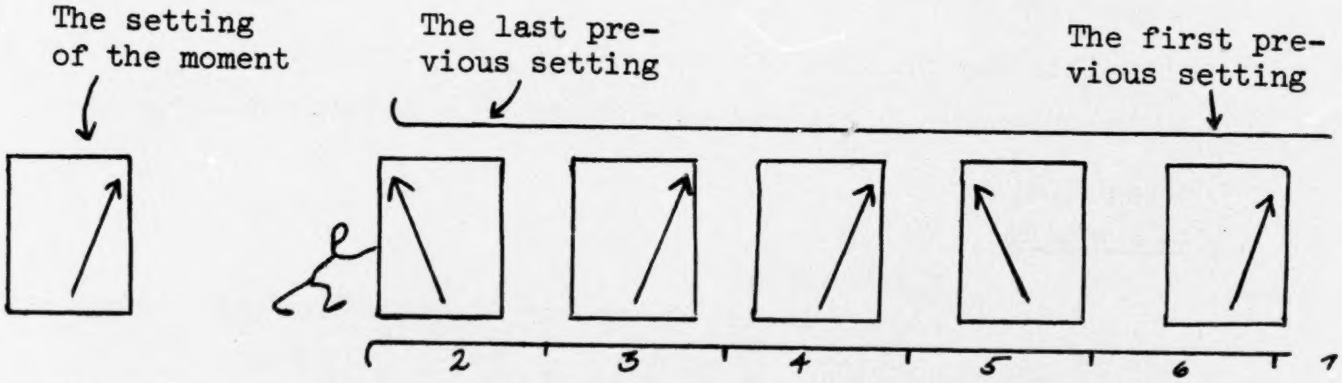
A condition code could be represented with a small square of cardboard to which a swinging arrow had been attached in such a way that it would point to either one of two possible conditions, success or failure:



or



Now let us require the computer to remember how the arrow pointed a moment ago, even though the current setting may have changed. This is certainly storable information, and we want to be able to retrieve it in reverse sequence. It is as if we were to stuff a whole series of remembered settings into a tube, and retrieve them according to the order in which they were stored away:


The setting of the moment
The last previous setting
The first previous setting
2
3
4
5
6
7

Along with each setting, its storage sequence number is noted. This number indicates the level or that particular setting. By letting the level determine whether or not a MATCH or FAIL will even be considered, a vastly expanded set of options is made available. This is the idea of block structure. It gives DISCUS a 2-dimensional aspect rather than one which is simply linear.

Every test carried out, whether it be dictated by a SCAN opcode or a TEST opcode, results in a condition-code setting. Action which follows (e.g., WRITE, SET, JUMP, etc.) may take place immediately, or it may be deferred until after one or more additional tests in the nest are performed, but in any case it can take place *only at the prescribed level preserved by the MATCH (or FAIL) which sanctioned it.*

How does one emerge from a deep level of operation? Simply through use of the END (or E) opcode. We pull the remembered settings out of the tube, as it were, and throw them away one by one, with an END statement for each preceding MATCH or FAIL statement in the nest. Sooner or later, back at Level 1, the total number of END statements in the text must equal the sum of MATCH and FAIL statements.

The following is an executable sequence*

```
SCAN;    }
MATCH;   }                      Level 1

  TEST;    }
  MATCH;   }                    Level 2

      TEST;    }
      MATCH;   }                Level 3

        SET;     }
        SCAN;    }              Level 4
        MATCH;   }

          WRITE;                Level 5

        END;                    Level 4

      END;                      Level 3

  END;                          Level 2

END;                            Level 1

Total MATCH + FAIL = 4
Total END          = 4
```

---

*As a convention for listing blocked coding, we sometimes use indentations resembling those employed in FOIL. It should be emphasized that they have no operative significance in DISCUS coding, however, but are intended only to help the reader visualize the block structure.

It is possible to jump out of a series of nested blocks at any point.

```
        SCAN;
        MATCH; }                    Level 1

          SET;
          SCAN;  }                  Level 2
          MATCH;

            TEST;
            MATCH; }                Level 3

              JUMP  (to some labelled location );    Level 4

           END;                     Level 3

         END;
         JUMP  (to some other labelled location);}   Level 2

      END;                          Level 1
```

The fact that execution in some cases never reaches the END statements at the far side of the block structure does not change the coding requirement: the END statements must always be there, and they must always equal in number the total number of MATCHes and FAILs used in the block. Thus in

```
MATCH;
JUMP     (to label address specified in operand);
END;
```

the END is indispensable.

While there is no restriction against jumping out of a nested block to a location encoded in such a way that it would be at a different level, care must be taken that the net effect is not to try to reach a level of less than 1. The total number of END statements encountered must not exceed the sum of the number of MATCH and FAIL statements passed-through, since going below level 1.

The current state of the condition code, as to whether it is set at success or failure, controls entry into a MATCH or FAIL block - either within the same nest or in another nest which has been JUMPed-to. When a block is successfully entered, the condition code that enabled the entry is preserved and a new level of condition code is brought into use. If no new TEST or SCAN is executed on this new level, the condition code as previously set is used. The next END statement in sequence brings the condition code to the next higher level in use, after saving the current code.

Example: (explanation below)

```
①     W   NAME THE THREE GRACES.;
②     A;
③     SC  AGLAIA;
④     M;
⑤         W    AGLAIA WAS FOUND;
⑥         SC   THALIA;
⑦         M;
⑧             W   THALIA WAS FOUND.;
⑨             SC  EUPHROSYNE;
⑩             M;
⑪                 W    EUPHROSYNE WAS FOUND.;
⑫                 J    XX;
⑬             E;
⑭         E;
⑮         W   WE ARE NOT SURE ABOUT EUPHROSYNE, BUT THALIA IS
              DEFINITELY MISSING.;
⑯ XX: E;
```

Explanation (Assume student answers "Aglaia, Clio, and Minerva.") Statement ③ looks for AGLAIA in the answer field, and since AGLAIA is found, the MATCH block starting at statement ④ is entered and the message at statement ⑤ is written on the screen. Since the scan for THALIA fails, the MATCH block starting at statement ⑦ is not entered, and execution continues at statement ⑭ , the END which corresponds to the termination of the MATCH block starting at ⑦ . The sentence in the operand field of statement ⑮ is then displayed.

Thus each MATCH or FAIL block is terminated with a corresponding END, and, if the block is not entered, all the contents of that block are skipped and execution proceeds to the corresponding END.

When a block is not entered because of condition codes not being favorable, all the nested statements within the by-passed block are ignored. Thus if a block on level 5 is encountered, but not entered because the condition code setting of the moment is failure while entry is contingent on match, or vice versa, and that block contains nested blocks at levels 6, 7, and 8, execution crosses over to level 5 beyond them, i.e., to the corresponding END statement.

BLOCK or B

There is one more OPCODE to be mentioned in connection with the DISCUS block structure, one of fairly recent development. The necessity for this new opcode - BLOCK - became evident only after finding that we needed a simple way of setting nested blocks apart as subroutines without too much concern as to where they occurred in the execution stream.

BLOCK simply stands in place of either MATCH or FAIL opcodes when the current condition code setting is immaterial and all that is desired is to bracket a section of code. (An example is given on page 78.)

The rule about END statements can now be restated: THE SUM OF ALL END STATEMENTS IN A BLOCK MUST EQUAL THE SUM OF ALL MATCH, FAIL, AND BLOCK STATEMENTS.

USE or U

A USE statement causes one or more logical instructions located elsewhere in the program to be executed as if they were actually listed at the location of the USE. The operand of a USE statement is always the single statement label address indicating the location of the instruction(s) to be executed. The label is not enclosed in quotes.

The scope of the USE depends on the nature of the statement bearing the access label specified in the USE operand:

a. If it is a FRAME statement (e.g., AAA:FRAME; or AAA:FR;)* then the entire frame is executed.

b. If it is a MATCH, FAIL, or BLOCK statement, then all commands at or below the entry point level will be executed, until the end of the block is reached.

c. All other statements are executed as individual entities.

As a corollary, execution returns to the point immediately after the USE statement when execution

---

*Discussed in next section.

a. The next FRAME statement, if a FRAME has been USED.

b. The first END statement encountered at the level at which the block was entered.

c. At the delimiter (;) of any other statement USED other than FRAME, MATCH, FAIL or BLOCK.

There will be numerous instances wherein the coder will want to use USE for bookkeeping operations. These are discussed in the section on useful subroutines in Part III.

The usefulness of USE can be exemplified briefly as follows:

a. Suppose the programmer has written a particular routine for inviting signoff, such as "You have been working for _____ minutes now, and your scoring rate is _____. Do you want to stop for awhile?" He might want to be able to call up this routine at several different points in the program, and he can do so by delimiting it with a BLOCK statement, the first of which he could label TIRED. He could then implant the following statement at several points in his program:

```
USE TIRED;
```

with the same effect as if he had repeated the entire routine at each location.

b. Suppose the programmer has written a testing routine to determine the "scoring rate" and to govern use of the frame described above, and for this purpose uses a block such as the following:

```
SCORCHK:  B;
              S   "STALLY" = "TTALLY" / "UTALLY";
              T   "STALLY > 7;
              M;
              U TIRED;
              E;
              E;
```

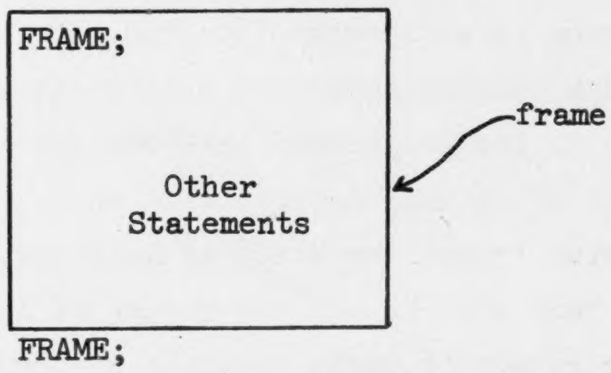
He needn't repeat this every time he wants to employ it in the program. USE SCORCHK; will suffice.

c. Suppose the programmer has prepared a lengthy WRITE statement (such as a carefully formatted list of categories).

He can effectively insert this single statement anywhere in his program simply by referring to its label in the operand of a USE statement; e.g. USE LIST22. UNLESS THE STATEMENT USED is a MATCH, FAIL, BLOCK, or FRAME statement, processing reverts immediately on reaching the first delimiting semicolon after the location cited in the USE operand.

Examples given following explanation of FRAME (next section) demonstrate some additional applications of USE.

FRAME or FR A FRAME statement marks the beginning of a segment of code. Also the word "frame" can mean the entire list of instructions (statements) beginning with a FRAME statement and ending at the next FRAME statement. For clarity we will indicate the word in this second sense by printing it always in lower case.



FRAME statements serve as reference point for entry or reentry into a particular sequence of code. They do very little processing, as such, and contain no operand. They may, however, be labelled and through such a label reached by means of a JUMP or USE from some other part of the program. When accessed in this or any other way they simply pass execution to the next statement in sequence.

The DISCUS compiler will cause a line of hyphens to be printed out in the object listing immediately ahead of the FRAME statement. (See example on page 24.) This is helpful in scanning and checking the program after it has been compiled.

A typical frame begins with a display of text followed by a directive that will elicit a response from the student.

Next follows an ANSWER statement, then one or more scanning and testing operations with their train of contingent actions: display of additional text and comment, scoring, branching to other routines, returning to the ANSWER statement, etc.

After all the operations specified in connection with that particular ANSWER statement have been encoded, and the coder is ready to go on to the next instructional step, he will probably decide to start a new frame (with a FRAME statement).

The frame - as a unit of instruction - is a teaching concept derived from earlier work in the design of programmed textbooks and teaching machines. Blocks of text, with multiple choices appended, were actually enclosed in a rectangular "frame," in the former. Below this figure appeared directions as to which page to turn to, depending on the student's choice of answer. Teaching machines concealed the correct answers mechanically until the student pushed a slide which carried his own answer irretrievably into the machine's insides.

Although computer-assisted instruction frees us from the constraints of the programmed textbook and its mechanical counterpart, it is only natural that an author/instructor should at first favor frame-like steps in developing his material. Later, the fact that he can use pieces of frames whenever he wishes, can return to whole families of frames at will, or jump from the middle of one frame to the beginning or middle of another, in due course will persuade him to structure his material more flexibly. After all, the frame as such does not govern the execution of the program; it is just one way of segmenting the flow of ideas.

A frame can be extremely short and simple, or so long and complicated that it involves hundreds of statements. If a definition is needed, one might say that a frame is a sequence of statements that contains one and only one ANSWER statement. This takes care of the kind of frame that makes no didactic pronouncement, but simply gives the student a chance to respond to something posed in a reaction WRITE in another frame (or in several other frames). It also covers the most intricate scan and test structures which follow the ANSWER statement before the next ANSWER statement occurs in the coding sequence.

This definition does not specify FRAME as a delimiter, because the FRAME STATEMENT IS PRIMARILY A LOCATOR. It has an important function in certain operations, but its presence or absence is not directly manifested. To explain:

FRAME can be thought of as the concierge of a small hotel, a hotel with wide-open windows, all on the ground floor. One can enter by way of a window and leave the same way, and the concierge will never be the wiser. If, however, one goes in by the front door, past the concierge, the latter unobtrusively makes a note of it and immediately notifies Headquarters.

Now let us say that the guest disappears - that is, the student signs off for the day. The record of his last known whereabouts is kept on file, and when he signs in again, Headquarters will see to it that that is where he materializes again.

Not to belabor the analogy further, let us say that we use a pointer system, and every time a FRAME statement is passed through, that location is recorded in a pointer and the record of the previous address is erased. The block level applying to the current operation is also recorded. Accordingly, the program knows where to resume execution after it has been interrupted, and at what condition code level. This is the RESTART FACILITY mentioned earlier.

FRAME fulfills still another function in connection with MATCH and FAIL counters. Whenever a FRAME statement is executed, all MATCH, FAIL and BLOCK counters are set to zero.

Now that the reader has been introduced to block structure, he can more readily appreciate the utility of the MATCH and FAIL counters of which we spoke earlier (page 68). Since these counters also work with BLOCK statements, we can refer to them as MATCH-FAIL-BLOCK or "MFB" counters.

In the following example, the statements which are <u>italicized</u> illustrate the operation of this device:

```
RUBENS: W WHAT TWO COLORS COMBINE TO MAKE ORANGE?;
        A;
        SC RED & YELLOW;
        M;
        W CORRECT;
        J TITIAN;
        E;
        SC RED;
        M 1;
        W RED IS CORRECT.; Provided the student doesn't get
                           both right in the first scan,
                           processing will test first for "red"
                           in the answer, and respond with
                           this, one time only.
        S "TALLY" = "TALLY" + 1;
        T "TALLY" = 2;
        M;
        W(ND) RED AND YELLOW ARE...(etc.);
        J TITIAN;
        E;
        J RUBENS;
        E;
        M;
        W YOU ALREADY SAID '"RED."'.;The next time around, and
                           thereafter, if the student answers
                           "red," this MATCH block will be
                           used instead.
        J RUBENS;
        E;
        SC YELLOW
```

(example continued on next page.)

```
        M 1;
        W YELLOW IS CORRECT.; Will be processed once only.
        S "TALLY" = "TALLY" + 1;
        T "TALLY" = 2;
        M;
        W(ND) RED AND YELLOW ARE...(etc.);
        J TITIAN;
        E;
        J RUBENS;
        E;
        M;          This MATCH will operate the second
                    time, and thereafter, after a
                    response of "yellow."
        W YOU ALREADY SAID '"YELLOW"';
        J RUBENS;
        E;
        F 2;        Response for the first two FAILS
        W NO, "ANSWER" IS INCORRECT. TRY ANOTHER COMBINATION.;
        E;
        F 1;        Response for the third FAIL.
        W SORRY, STILL WRONG. BOTH HAPPEN TO BE
          SO-CALLED '"WARM"' COLORS. TRY ONCE
          MORE.;
        J RUBENS;
        E;          Response for fourth FAIL.
        W YOU SHOULD HAVE ANSWERED '"RED AND YELLOW."';
TITIAN: WHAT ARE THE '"PRIMARY"' COLORS?;

                  etc.
```

The numeric expression used as an MFB counter doesn't have to be a simple integer; it can be an expression such as "N" + 4 - "Y". In such a case the system makes the necessary retrievals and/or computations to establish the actual number of times the M or F or B statement will be allowed to operate.

NOTE or N Oftentimes the programmer will want to make a permanent or semi-permanent memorandum of his reasons for coding a sequence in a particular way, or to indicate a transition, or to relate sections of code to topical headings in the author/instructor's outline. He wants these memoranda to reappear automatically in the program listing whenever it is recompiled, without, however, affecting execution in any way.

The NOTE statement does this. It is completely inert as far as the program is concerned, and its only function is to preserve and reiterate, in the OBJECT listings, whatever the programmer has placed in its operand.

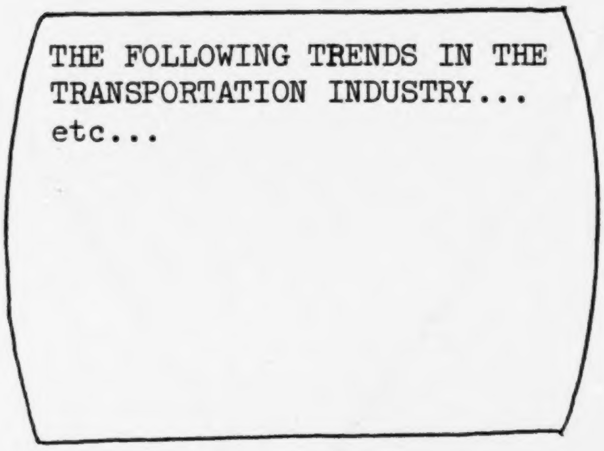
Statement	N STARTING DRILL AND PRACTICE IN PARSING;
Result	N STARTING DRILL AND PRACTICE IN PARSING; is printed in the OBJECT listing, in its proper location, with a statement number assigned.

NOTE is seldom labelled, because there is not much point in accessing a statement that does nothing. Occasionally, however, there arises a need to assign two labels to a single statement. Since current implementation does not permit this, one of the labels can be assigned to a NOTE statement, which is placed immediately ahead of the statement in question. On execution, if that label is JUMPed to, processing simply goes on to the next statement.

Statements

```
                  .
                  .
                  .
                J REVIEW37;
                  .
                  .
                  .
                J DICTUM37;
                  .
                  .
                  .
      REVIEW37: N;
      DICTUM37: W THE FOLLOWING TRENDS IN THE TRANS-
                  PORTATION INDUSTRY...etc...;
```

Result (of either JUMP)



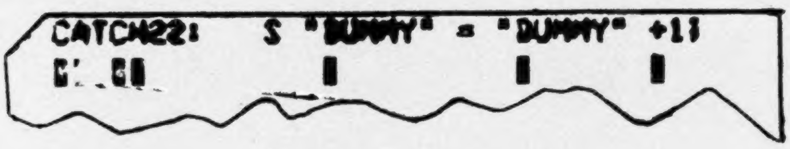
```
THE FOLLOWING TRENDS IN THE
TRANSPORTATION INDUSTRY...
etc...
```

NOTE can also be used temporarily to suppress a statement (together with all statements which depend on it). This is sometimes desirable when the programmer wishes to enter certain items in the program but to keep them in abeyance as far as execution is concerned. He can imbed this material in his source deck and see it printed out in the OBJECT listing, and yet be sure it won't be executed, simply by adding NOTE or N ahead of the statement to be suppressed (but following its label, if any). This has the effect of turning the opcode of that statement into operand material, which will be (because of the N opcode) completely disregarded during execution.

When the programmer is ready to activate the statement in a subsequent compilation, he simply removes the N opcode.

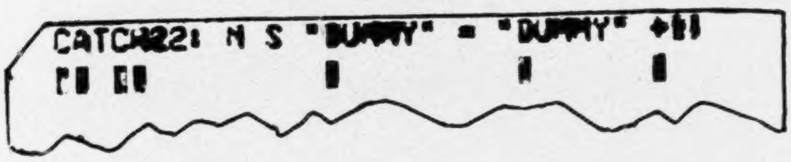
The device is made the more convenient if statements are always punched with a few extra columns preceding the (normal) opcode, in addition to those needed for labels.

Statement (as normally punched)



```
CATCH22:   S "DUMMY" = "DUMMY" +1;
```

(execution suppressed)



```
CATCH22: N S "DUMMY" = "DUMMY" +1;
```

PART III - PROGRAMMING IN DISCUS

IDENTIFYING THE PROGRAMMER

The functions performed by the several DISCUS statements as described in PART II need to be thought of as tools by means of which the craftsman may be able to shape satisfactory computer-assisted dialogue. Like any precision tools, they need to be used deftly, and in a number of different combinations, in order to achieve a desired result.

At this point we should pause to consider who will be using these tools. Will he be a professional programmer, one already trained in some standard computer language, perhaps several? Will he undertake DISCUS programming simply as an extension of his repertoire?

Or will he, instead, be a person whose main commitment is to education or training in an academic, commercial, or industrial environment? If so, will the production of course materials for direct implementation be his principal concern, or will he be oriented more toward research and development in the educational and training processes themselves?

Will he be a specialist in some subject field, one who works in an educational environment, perhaps, but whose time and energies are largely preoccupied by the demands of his specialty? Might such a person take up CAI programming in order to promulgate his specialty more effectively?

Might the DISCUS user be one whose interests and duties combine elements of all the foregoing in a pattern that is unique to him alone?

In trying to identify and characterize the CAI programmer in standard terms, we find that no one set of qualities, interests, and objectives suffices. Between the extremes of the systems programmer, on the one hand, and the preoccupied scholar, on the other, there will be many who will find CAI programming a useful adjunct to their other talents. In addition there will be some who will take it up as their exclusive activity. We expect that the latter will cluster largely in the educational specialist sector, but this may not always hold true.

It is difficult to conceive of an individual at either extreme who would be both willing and able to produce, unaided, instructional material of suitable quality. In "The CAI Author/Instructor", Meredith discusses the effect on CAI of the essential dichotomy between educational substance and educational tools. He postulates a role for the subject specialist wherein that individual need not master the technicalities of computer programming. By implication the CAI "programmer/instructor" would be the other half of a pair of specialists sharing a single educational task.

Whether one accepts this approach, or adheres to the belief that one should merge all functions of the author/instructor and of the programmer/instructor in a single person, the functions themselves can be considered separately, and in the following pages we speak of them in this sense - as functions. We personify them separately, for the sake of clarity and convenience, but not to imply that only a team approach will work. The reader can decide for himself whether he wants to be both members of the team. Manifestly this Manual is limited to dealing with the programming function except insofar as it is necessary to speak of its relationship to the authorship function.

CAI AS A PRODUCT

We have spoken of DISCUS statements as precision tools which the CAI programmer uses to shape satisfactory computer-assisted dialog. The product, in his case is nothing as graphic and self-evident as a watch or a painting or even a piece of music. The programmer deals with ideas and processes in a heuristic way; the results are not immediately evident. His product is only a plan of idea-transitions; it is protean and probablistic, a plan of happenings many of which may never happen.

SATISFACTORY COMPUTER ASSISTED DIALOG

What is meant by "satisfactory computer-assisted dialog?" Satisfactory to whom?

In a CAI context, we feel that both the student and the author/instructor need to feel that they are well served by the computer as a means of communication between them, bearing in mind that disjunctures of time and place might otherwise prevent communication altogether. The author/instructor needs to be reassured that his statements will not - in the convolutions of the dialog - somehow be turned topsy-turvy. The student needs to be able to convince himself that somewhere a human, however remote, is attempting to reach him as an individual. Both need to be willing to forego the exquisite controls and balances which govern the flow of live conversation, and which on reflection they would be unwilling to endow a machine with in any case.

Computer-assisted dialog is "directed dialog," with almost all of the direction in the hands of the author/instructor and the programer/instructor. This in itself is not a bad thing: students are used to being lectured at, not with, and even seminars have a topic of some kind, and a leader. In highly interactive CAI the sharing of some measure of direction is entirely feasible, and eventually - as programs increase in number, versatility, and general availability, perhaps to the extent of coalescing into huge learning complexes - much of the supervision of directed dialog will pass to the person seated at the student console.

PROGRAMMING AS AN EXERCISE IN ANONYMITY

We have purposely omitted the programming function from the above criterion of satisfactory computer-assisted dialog, because the programming function facilitates, rather than judges, the final result. He is a communicator, rather than a communicant. He has the responsibility of determining how well the author/instructor and the student are served, according to the system's capabilities as he sees them. He decides not only whether a certain thing can be done, but also whether it will be useful enough to either of his patrons to make it worth doing.

There is, after all, a limit to available programming effort. The time spent in making one frame super-elegant may lead to the slighting of others, to the detriment of the sequence as a whole. On the other hand, it may lead to the discovery of contrivances which will be useful in a number of frames. Accordingly, when a programmer devotes many hours to a particular subroutine for a particular frame, he has, or should have, some expectation of being able to use it in other frames as well, with adjustments that will be minor and preferably automatic.

As far as the author/instructor is concerned, the programmer speaks for the system hardware, offering a mixed collection of services. These are usually far greater in scope than the author/instructor who is new to the field will have envisaged or will be prepared to utilize. It is essential that they not be dragged in just for the sake of using them as toys. On the other hand, the author/instructor should be reminded from time to time that they are there.

As far as the student is concerned, system mechanisms should be as unobtrusive as possible, in order to permit the level of communication we seek. Whatever the system conveys to him is conveyed on behalf of the author/instructor. The programmer, as part of the system, keeps well out of sight. He avoids imposing more than an absolute minimum of housekeeping chores on the student. He busies himself between the two principals, a collaborator of the one and an observer of the other. He is there, but like the ne'er-do-well father who shows up unexpectedly at his son's fine wedding, he shouldn't rush in and shake hands all around.

PREPARATIONS A level of mutual understanding should join the author/instructor and the programmer/instructor, so that the former will have a good grasp of system capabilities and the latter will have a good grasp of the subject to be taught. Before the actual writing of a program is begun, a number of parameters need to be specified, or at least described closely enough so that collaborators will not find themselves drifting apart later on.

It is best that this be done in writing. No informal understanding, however agreeable to both parties, can properly substitute for written instructions. Even if both functions are undertaken by a single individual, he should carefully profile his intended student set, course objectives, subject matter, and strategy - in advance and in writing. The following checklist may be useful in this connection:

Define the course objectives.

What is expected to be the students' median age?
" " " " " " " *academic level?*
" " " " " " " *preparation and/or screening?*

How is the subject matter conventionally organized and presented?

What are its boundaries?

What is the nature of adjacent subject matter, and how permeable are the inter-subject boundaries?

The general plan of organization. Will the instruction be sandwiched between lectures or labs?

What will be the policy on the handling of requests for review. Should review consist of iteration, restatement, compression, or a combination of all three?

What will be the policy on the handling of requests beyond the scope or depth of the "involuntary stream." If voluntary digressions are permitted, where and how are they to be brought back to the mainstream?

What will be the policy governing sequencing of topics or the resumption of topics on "restart" (i.e., when the student signs back onto the terminal after an interruption)?

What elements are to be the subject of scoring?

What statistical data need to be accumulated?

CORPUS

Next, the author/instructor should furnish the programmer/instructor (or the author/instructor/programmer should furnish himself) with a body of substantive instructional material (corpus) which - if not covering the entire course - will at least be complete and well polished as far as it goes:

A list of textual statements to be used.

A list of questions or requirements designed to elicit responses from the student.

A list of anticipated responses according to key words (e.g., a typical response with key word(s) underlined.)

A list of conditional rejoinders (not necessarily complete, because not all reasonable responses can be imagined by the person who generates the text; the programmer himself should be able to suggest additional possibilities.)

This is the heart of the dialog. Its preparation is somewhat more demanding than would be the writing of an effective, publishable textbook. It should be as explicit as possible before the programmer picks up his tools. There will be revisions, of course, after the author/instructor develops an ear for computer-assisted dialog. Hopefully some of the initial frames can be implemented in the system for just this purpose. But without written bases as points of departure for such revisions, the programmer will find himself floundering in a morass of "Who said what? Where am I?"

- - -

In order for the system to serve both educator and student satisfactorily, certain favorable conditions must be established at the terminal:

The student needs to learn to use the equipment as quickly and with as little fuss as possible. After being told how to sign on and off and how to send his responses to the computer, he should - insofar as possible - be freed of the mechanical imperatives which tend to intrude between him and the course material.

There appear to be two levels of instruction involved here: instruction on how to use the terminal, as above, and instruction on how to formulate input. An example of the former would be the imparting of the method of backspacing over mistakes, something the student should be told by pre-instruction. The latter represents some inherent constraint or artificiality in the program itself, such as might prevent input from being "understood." To illustrate this, suppose we had the following display:

THE "STATUTE OF FRAUDS" IS PRIMARILY CONCERNED WITH

1. TRANSACTIONS IN JEWELRY
2. STOCK AND BOND SALES
3. REAL ESTATE TRANSFERS AND LEASES
4. THE 'USED CAR RACKET'

___

Suppose that for some reason the system or the CAI language in use, or both, imposed one or more of the following rules:

ANSWER ONLY WITH NUMERALS
ANSWER ONLY WITH NUMERALS OR A SINGLE KEYWORD
ANSWER WITHOUT ANY SPELLING ERRORS
ANSWER WITHOUT REVISING SYNTAX IN ANY WAY
DON'T SUBSTITUTE 'THE FIRST' or 'ONE' for '1', etc.
DON'T USE DOUBLE BLANKS
DON'T ANSWER WITH MORE THAN ONE OF THE CHOICES
DON'T USE MORE THAN 8 CHARACTERS IN YOUR ANSWER
DON'T COMPLAIN OR ASK QUESTIONS
DON'T . . .etc.

How much of this sort of thing should be inflicted on the student before he has a chance to answer the question? The best solution is to eliminate the constraint, but there will be times when the programmer will feel that the likelihood of deviant response in a particular frame is so slim that it is not worth the labor of accommodating it. It appears to us that this sort of decision should be reflected in post-instruction. There is no point in telling the student ANSWER WITH NUMERALS ONLY if that is what he was going to do anyway. The system intrudes much less if it corrects and instructs only when an item of student input shows that correction and instruction is necessary.

The choice between pre-instruction and post-instruction depends on the exigencies of each particular frame, but the norms influencing these choices should be agreed in advance between the author/instructor and the programmer. If the latter works in a CAI language as agile as DISCUS, the policy can in fact be one which abjures pre-instruction altogether, or almost altogether. One can, in other words, eliminate from the phrase "directed dialogue" the connotation of directing the form in which the student answers must be couched in order to be processed by the system.

- - -

The presentation of instructional material in ways that permit the student to take an active part in his own learning process is an art in itself. It is not enough to pause every few sentences to ask a question merely to find out if the student has read and understood what he has been told; the student should be enabled to extend or exemplify the concept in some small way. This means stretching the system to the limit in order to cope with a very broad range of responses. When the types of response elicited by the author/instructor threaten to exceed this limit, e.g., when the proportion of unanticipated, or fail-match responses would be excessively high, he should be so advised.

It is useful to categorize possible student answers along the following lines as a basis for assessing the performance of a given set of SCAN specifications:

A_L	(Answer, Legal)	A string of student input of limited length, containing at least one term relevant to the question or its context, and in grammatical order.
A_I	(Answer, Illegal)	Any string of student input not meeting the above requirement.
A_{LP}	(Answer, Legal, Perceived	Any A_L which is predicted as possible input.

A_{LPM}	(Answer, Legal, Perceived, Matched)	Any A_{LP} which the author/instructor chooses to deal with specifically.
A_{LPF}	(Answer, Legal, Perceived, Failed)	Any A_{LP} which the author deems so remote or so inconsequential or so difficult to deal with uniquely that it should be failed.
A_{LU}	(Answer, Legal, Unperceived)	Any A_L not perceived as a possibility.
F	(The sum of failed Answers)	$A_I + A_{LU} + A_{LPF}$

The type of answer which principally occupies our attention is A_{LPM}. It is not necessary that the author or programmer have in mind everything that the student might say embodying the A_{LPM} element(s); it _is_ necessary that he select some elements which are

> likely to match with a broad range of responses relevant to a question,
>
> will match members of semantically similar input, and
>
> will distinguish semantically dissimilar input by failing to match with it.

The range of A_{LPM} is directly influenced by the form of the questions which precede them. A question posing a very rigid requirement (e.g., True-False) reduces the number of A_{LPM} to a minimum, but at the same time it vitiates the conversational mode. The goal should be to open up questions and to give the student as much freedom in formulating his responses as we can, without letting match probability fall below acceptable levels.

Part of the author/instructor's task is to define, or at least coherently to describe, the A_{LP} which he wishes to intercept. The SCANs that are to accomplish this call for close collaborative effort on the part of both the author/instructor and the programmer, the former contributing most (but not all) of the lexical strategy, and the latter contributing most (but not all) of the mechanical tactic.

In the selection of a single keyword to be specified in a SCAN statement, we naturally want to choose one which will, if present, almost certainly indicate that the student has expressed a particular meaning to which we can respond unequivocally. If there appears to be excessive risk in relying on a single keyword, i.e., if the word might reasonably be used by the student in a way different from that to which we want to respond, we try to restrict it somehow by combining it with one or more additional words. While this increases the confidence level in respect to the program's understanding of that which it intercepts, it decreases the versatility of that one SCAN parameter insofar as intercepting variant expressions of the same meaning is concerned.

This effect approaches the absolute when one specifies the key-string as a literal, ruling out "match" if the student inserts or omits anything at all (even an extra blank or two) between the specified elements. Thus

SC *'RED, WHITE, AND BLUE'*;

would not intercept

red, white, pink and blue

or

red, white, blue

or even

red, white and blue

Such rigidity may be exactly what is wanted sometimes, of course, but more commonly literals are used to specify parts of keywords rather than longer strings, their function being to afford some latitude in spelling, syntax, etc. Thus

SC *' STOR'*;

accepts storage, storing, stores, storable, store, etc. (and stork, stormy, Storting too, but these are remote enough to be acceptable risks.)

Literals aside, it might be said that for each word specified in conjunction with a keyword, the likelihood of misinterpretation decreases very rapidly, whereas the likelihood of missing identical or nearly identical meanings increases by a similar factor. But this should not be construed other than as a gross representation of a whole series of effects and counter-effects tied to the probabilities of the English language. Input varies in length, and one might ask to what extent a long input string would be more likely to contain a given key-string than a short one. Perhaps none at all.

Not all associative words used with keywords carry the same discriminatory power. In fact, one finds that sometimes the words which are the most obviously associative carry no more certitude of correct interpretation when combined with the keyword than do those which (syntactically, at least) appear to be more remote yet which encompass a better range of variance. Thus it is possible to expand the key-string with elements which will greatly strengthen it, without entailing loss of versatility in anything like the same ratio.

Fortunately, one need not rely on a single keyword of key-string to intercept a single meaning: the use of "or-connected" suboperands in the SCAN statement permits a diverse approach, with each suboperand attacking the problem from a different angle. (Presumably the program reaction to any one of a group of "or"-ed suboperands in a single statement is intended to be the same. Otherwise, they should be set up in separate SCAN statements and dealt with uniquely.)

The "and"-connected suboperand is really not much different from the multiple-word, or key-string, operand discussed above, the only difference being that its elements will match input of the same elements in any order. Where

SCAN *RED WHITE BLUE;*

will match input containing those three words in that order (e.g., roses are red, snow is white, and violets are blue) it will not match input containing them in any other order (e.g., I have a blue hat with red and white streamers.) But

SCAN *RED & WHITE & BLUE*; will.

"And"ed suboperands are more rigid than single keywords, but obviously less rigid than ordered keystrings, because they do permit inversions and permutations. Since content rather than form is usually the chief desideratum, this kind of SCAN becomes quite useful, even if at the time it is chosen neither the author nor the programmer have in mind all the forms it might intercept.

Another feature of the "and"ed suboperand is that it allows one to negate any parameter without negating the whole, as happens in a simple key-string if one assigned ¬ to any of its parts.

Statement	SCAN ¬ *OSCAR STRAUSS*; or SCAN *OSCAR* ¬ *STRAUSS*
Result	The "not"-sign negates both Oscar and Strauss.

If we merely wanted to intercept any Strauss other than Oscar, we would specify

SCAN ¬ *OSCAR & STRAUSS*; or
SCAN *STRAUSS &* ¬ *OSCAR*;

"And"ed and "or"ed suboperands let us specify SCAN parameters with great precision without forfeiting the degree of open-ness, or flexibility, which we require in a given situation. Coupled with the "not" facility (¬) and the use of stepped or conditional SCANS we find that we can interpret student input very accurately indeed.

In the absence of a "not" facility one can accomplish some of the same effects by erecting screens which scan for unwanted elements first, before scanning for wanted elements. Using the OSCAR STRAUSS example, above, one would scan separately for OSCAR, ahead of the STRAUSS scan, and if the former is found, jump over the latter.

```
          SC OSCAR;
          M;
          J  WALTZ;
          E;
          SC STRAUSS;
          M;
          W  YES.;
          J  NEXTFR;
          E;
WALTZ:    SC STRAUSS;
          M;
          W  NO, NOT OSCAR STRAUSS.;
          J  NEXTFR;
          E;
```

There are two ways in which DISCUS permits us to improve on this, Ⓐ one using the "not" facility (for the case where we don't want to make a point of telling the student that Oscar is wrong), the other Ⓑ using stepped SCANS:

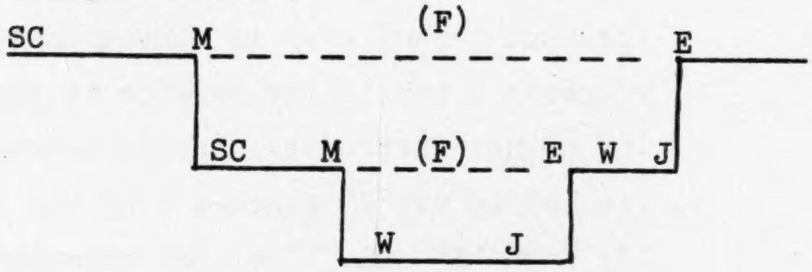
Ⓐ

```
SC¬OSCAR & STRAUSS;
M;
W  YES.;
J  NEXTFR;
E;
```

Ⓑ

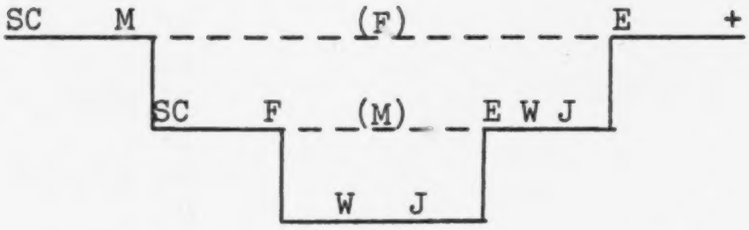
```
SC STRAUSS;
M;
SC OSCAR;
M;
W  NO, NOT OSCAR.;
J  NEXTFR;
E;
W  YES;
J  NEXTFR;
E;
```

(diagrammed as:)



B represents an extremely effective way of dealing with faulty answers, because it can cover errors of omission as well as errors of commission, simply by using a FAIL block instead of a MATCH block. Suppose we wanted OSCAR STRAUSS:

```
SC STRAUSS
M;
SC OSCAR
F;
W  YOU DIDN'T SPECIFY OSCAR
   STRAUSS.
J  NEXTFR
E;
W  CORRECT':' OSCAR STRAUSS.
J  NEXTFR
E;
```



Block structure moreover permits one to scan for the same word more than once in the same frame without the constraint that the first such SCAN is the only one that will succeed.

As indicated earlier, TEST and SCAN are much alike. Both govern decisions in block structures, but whereas TEST may be satisfied according to its terms by a logical relationship (>, ¬ , =, or <) of the whole variable tested, SCAN requires equality, but only in the part specified. Also while the contents of any variable may be TESTed, only the contents of the ANSWER variable may be SCANned.

TEST is relied on for most of the housekeeping chores involved in careful CAI programming. Often such functions can be standardized and tucked away in subroutines, callable by the USE command wherever a particular service is required. For example, suppose the author/instructor wanted to invoke review sequences when "X" reached 10 or "Y" reached 7 or the two together amounted to over 13; both "X" and "Y" to be incremented whenever the student made certain types of errors. The subroutine to accomplish this should be USEd at the beginning of each frame, which is the logical breakaway point for any review. (Note that we use a block structure to USE several statements together.)*

---

*Assume "X", "Y", and "Z" have been defined as arithmetic variables, and REVY is the label of a review sequence.

```
          FR;
          U REVIEWA;
          W (ongoing text);
          =========================
REVIEWA:  B;
             T "X" > = 10;
             M;
                U NOTIFY;
                J REVY;
             E;
             T "Y" > = 7;
             M;
                U NOTIFY;
                J REVY;
             E;
             S "Z" = "X" + "Y";
             T "Z" > = 13;
             M;
                U NOTIFY;
                J REVY;
             E;
NOTIFY:      W I THINK A SHORT REVIEW WOULD
               BE GOOD AT THIS POINT.;
             E;
```

(Observe that the statement labelled NOTIFY never executes sequentially, because it follows a JUMP at the same level. An alternate treatment would be to substitute it for one of the U NOTIFY statements.)

Suppose the author wanted to prevent repeating exactly the same review sequence (REVY, in the example) in the case of a student who continues to have difficulty. One solution would be to furnish a control along the following lines: (Added statements are indicated by arrows).*

```
REVIEWB:  B;
             T "X"> = 10;
             M;
                U NOTIFY;
                T "REVYX" = 1;
                M;
                   J REVY2;
                E;
                S "REVYX"= 1;
                J REVY;
             E;
             T "X"> = 7;
             M;
                U NOTIFY;
                T "REVYX" = 1;
                M;
                   J REVY2;
                E;
                S "REVYX" = 1;
                J REVY;
             E;
             S "Z" = "X" + "Y";
             T "Z" > = 13;
             M
                U NOTIFY;
                T "REVYX" = 1;
                M;
```

---

* Assume "X", "Y", and "Z" and "REVYX" have been defined as arithmetic variables.
(REVY and REVY2 are labels of the two review sequences.)

```
                     J REVY2;
                  E;
                  S "REVYX" = 1;
                  J REVY;
     NOTIFY:      W I THINK A SHORT REVIEW WOULD
                    BE GOOD AT THIS POINT.;
               E;
            E;
```

It should immediately be apparent that the added statements comprise three identical groups, which might be written as a separate subroutine (or 'sub-subroutine' if you like) and USEd when needed:

```
   REVYZ:  B;
               T "REVYX" = 1;
               M;
                  J REVY2;
               E;
               S "REVYX" = 1;
           E;
```

USEable subroutine block

```
               FR;
               U REVIEWA;
               W (ongoing text);
          ===========================
    REVIEWA:  B;
                  T "X > = 10;
                  M;
                     U NOTIFY;
                     U REVYZ;
                     J REVY;
                  E;
                  T "Y"> = 7;
                  M;
                     U NOTIFY:
                     U REVYZ;
                     J REVY;
                  E;
                  S "Z" = "X" + "Y"
                  T "Z" > = 13;
                  M;
                     U NOTIFY;
                     U REVYZ;
                     J REVY;
NOTIFY:              W I THINK A SHORT REVIEW WOULD BE
                     GOOD AT THIS POINT;

                  E;
               E;
```

The block of seven statements can be stored away anywhere in the program where it will be protected against accidental execution, e.g., after a JUMP. Another appropriate location is any one of the points at which the coder wants to provide for its use, where it will naturally take the place of the USE command, at the same time it continues to be USEable from other points in the program.

In the above, we have reduced the fifteen statements required in the original augmentation (page 91) to ten, and have provided a subroutine module that will possibly be useful in some other context, because REVYZ is not bound to REVIEWA even though we happened to write it as part of that subroutine. It can be brought in at any level above the very lowest (250) and function quite properly.

It is axiomatic in programming that one should avoid repetitive code whenever possible, i.e., when a subroutine can be devised that will always and invariably achieve a certain effect when called upon. By being alert to the use of subroutines, the programmer saves himself considerable drudgery, besides reducing the chance of random error (in coding and keypunching) to which repetitive coding is susceptible. His program will compile faster and occupy less storage space. Execution time will be increased, but this is usually a minor point.

To return to the problem for which the REVIEWA* subroutine was suggested as one solution, namely the situation wherein the author wants to invoke review sequences whenever the values in certain variables reach prescribed levels:

A second method would be to put TESTs at the head of each set of review frames, and poll them in a series of JUMPs. The difficulty is that it would be impossible for execution to return to the point whence it was diverted, since JUMP is an unconditional command. Only USE provides return to the point of origin on completion of the frame, block, or statement that is USEd.

Third, it is possible to nest whole series of frames in blocks, entry to which depends on the results of TESTs and SCANs, and this might be a good way to set aside complicated review or enrichment routines whose execution would automatically adjust to combinations of values in a set of variables. An example of this kind of development is not attempted here, however, because it would involve excessive supporting detail of no immediate interest.

---

*These labels are all arbitrary and have no special meaning in DISCUS.

It may be asked, in connection with the subroutine REVYZ, worked out on page 103, why we needed a subroutine at all, when by simply limiting the number of times a MATCH, FAIL, or BLOCK block may be entered we might prevent the same review sequence (REVY) from being executed twice. Why not encode REVIEWA as follows?

```
REVIEWA:  B;
             T "X" >= 10;
             M 1;
                U NOTIFY;
                J REVY;
             E;
             M;
                U NOTIFY;
                J REVY2;
             E;
             T "Y" >= 7;
             M 1;
                U NOTIFY;
                J REVY;
             E;
             M;
                U NOTIFY;
                J REVY2;
             E;
             S "Z" = "X" + "Y";
             T "Z" >= 13;
             M 1;
                U NOTIFY;
                J REVY;
NOTIFY:         W I THINK A SHORT REVIEW WOULD BE GOOD AT
                  THIS POINT.;
                E;
         E;
```

1. Immediately following each ANSWER statement, insert

```
USE UTILITY;
```

2. In a protected location, establish a block of code beginning with

```
UTILITY:  B;
```

and ending with

```
UTILANS:  A;
          E;
```

3. Inside the UTILITY block, a series of subroutines may be nested as separate modules, to be added to or changed or removed as the coding progresses and the exact nature of the service required is more fully perceived by the programmer.

a. To help the student sign off:

```
B;
SC SIGN OFF, QUIT, TIRED, 'FINI', 'TERMIN'
   SPLIT, GET OUT, 'M THROUGH', 'M THRU', LOG;
M;
W  IF YOU WANT TO SIGN OFF NOW, JUST
   TYPE '"EXIT"';
J  UTILANS;
E;
```

(1) "EXIT" happens to be the sign-off convention in use under ILR Berkeley TMS Monitor. At ILR UCLA, the equivalent term is "END".

b. To display the previous statement (often desirable when the student has made two or three unsuccessful stabs at a question). An arithmetic variable, previously established, will have been incremented every time a FRAME statement is passed through, e.g., SET *"FRCOUNT" = "FRCOUNT" + 1;)*

```
B;
SC REPEAT & PREVIOUS, DISPLAY & PREVIOUS,
   CHOICES, AGAIN, 'FORG' STATEMENT,
   'FORG' LIST, SHOW ME, ME SEE, WANT
   SEE;
```

There are two errors in this treatment:

One is that it does not prevent the student from qualifying for REVY two times or even three times in a row on the different criteria. That is, he could have a total Z of 13 and be jumped to REVY, then a total X of 10 and be jumped again to REVY, then a total Y of 7 and again be jumped back to the same old review. There is no way of grouping the three independent tests in such a way that a single limiting MATCH-FAIL-BLOCK-counter will monitor them unless we set up a fourth block to test a flag dedicated to this one function - which brings us right back to the REVYZ routine (p. 103).

The other difficulty is that since MFB-counters are reset every time execution passes through a FRAME statement, the device would work only in the unlikely event that REVY contained no FRs, and then only for the duration of the current frame in the mainstream of the instruction.

We can draw a general conclusion from this, namely that the MFB-counters are not particularly suited to subroutines intended for inter-frame use except for operations strictly internal to the subroutines themselves.

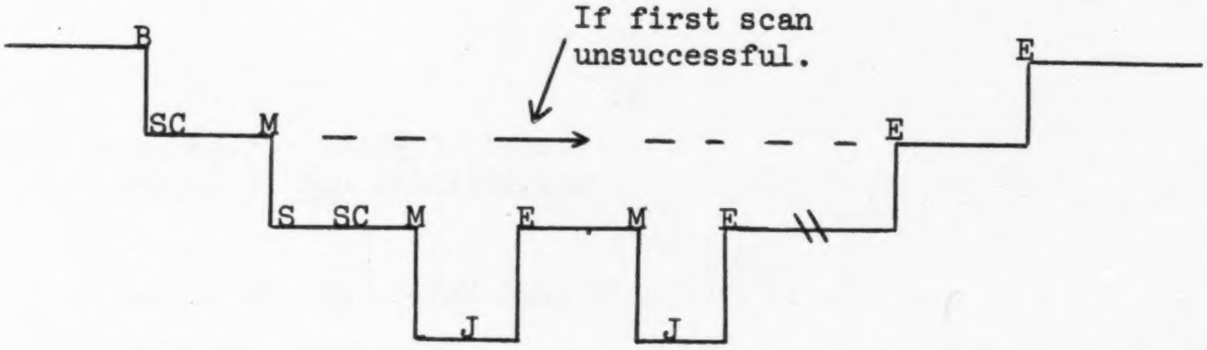
A characteristic which limits a device in one application (as above) may be turned to advantage in another. For example, one might want to reset MFB-counters during execution of a frame, under certain conditions, without having to leave that frame and then return. This can be effected by inserting a frame statement (FR;) - which need not be labelled - at any point where the MFB-counters are to be reset to zero.

- - -

Before the programmer starts writing frames, he will be well advised to consider the kinds of utility services that should be available more or less continuously during the running of the program - services for both the student's and his own benefit. DISCUS users may be able to adapt to their needs. They should be thought of as if they were recipes in a cookbook, which anyone is welcome to try.

```
M;
S "ANSWER" = "FRCOUNT';
SC 1;
M;
J  (label of first frame of main text);
E;
SC 2;
M;
J  (label of second frame of main text);
E;
.
.
.
E;
```

diagrammed as follows:



B
If first scan unsuccessful.
E
SC
M
E
S
SC
M
E
M
E
J
J

c. To allow student to go back to previous frame:

```
B;
SC GO BACK, PREVIOUS, REPEAT, PRECEDING,
   REVIEW, LAST;
M;
S  "BACKUP" = "FRCOUNT' - 1;
S  "ANSWER" = "BACKUP";
U  WHERE;
E;
E;
```

d. To allow author/instructor, programmer, editor, or "debugger" to check the status of variables during processing:

```
                B;
                SC EDITOR
                M;
                W  EDITOR INFORMATION':'/;
                W(ND) ƀ_____________ƀ=ƀ"_____________";
     (Name of variable, not            (Name of same
              in quotes)                variable, in
                                        double quotes)

                          etc.

                E;
                E;
```

e. PROCTOR MODE, to allow programmer, etc., to jump execution to another part of the program without using DISCUS system author mode

```
                B;
                SC  GO TO;
                M;
                SC (label of FRAME statement at desired
                   destination, not in quotes.);
                M;
                J  (same label, not in quotes.);
                .
                .
                .
                E;
                W  SORRY, UNABLE TO COMPLY.;*
                E;
```

This routine allows one to access topical subdivisions of the program according to an outline of the course, keyed with program labels corresponding to its various rubrics. It is not as precise a method of jumping around in the program as the //F=t command affords (see p. 137) but is somewhat more convenient for the non-programmer.)

---

* This takes care of the unfound label, and of the student who says "go to hell.")

f. To gather statistical information about student input, e.g., common spelling errors

```
B;
SC (word); (or SC (word), (word);
               SC (word) & (word);
               SC (word) (word); etc.)
M;
S "WORDA" = "WORDA" + 1;
E;
SC (word);  (etc.)
M;
S "WORDB" = "WORDB" + 1;
E;
.
.
.
E;
```

STARRED VARIABLES An unusual feature of DISCUS is an operation that can be performed *on* a character variable by placing it between two asterisks in a SCAN operand, thus

```
SC        '␢'*"VIRIDIUM"*'␢'      ;
```

Instead of the contents of the variable being *scanned*, as would be the case if the statement were written in the usual way:

```
SC        "VIRIDIUM"              ;
```

the variable is filled with data from the ANSWER FIELD.

In order for this to happen, a match must occur between something in the answer field and SCAN elements immediately preceding and following the starred variable.

At present only character variables may be used.

```
     CAT:      D(C) 50;
     W         (something);
     A;
     SC        ' S' *"CATS"* 'S ';
```

The SCAN, operating on an answer field of 'SLIVERS', for instance, would obtain a match, and would most assuredly put LIVER into "CAT".

```
     SC  ' '  ' ' *"CAT"* 'S ';
```

operating on an answer field of 'ARGYLE SOCKS' puts SOCK into "CAT".

```
     SC        'S' *"CAT"* ' ';
```

operating on the same answer field puts OCKS into "CAT".

In single operands or in ored suboperands, only the first suitable ANSWER field gets inserted, because the SCAN operation terminates immediately on success. Thus

```
     SC        ' S' *"CAT"* ' ';
```

operating on an answer field of 'SALLY SELLS SEA SHELLS BY THE SEA SHORE' matches immediately with b̸SALLYb̸ and only ALLY gets put into "CAT", not ALLY HELLS EA HORE.

PART IV - CONCISE DISCUS SPECIFICATIONS

SPECIFICATIONS This section of the MANUAL defines the basic elements of DISCUS, for use as ready reference and as a recapitulation of the material presented in PART II and PART III. As stated in the INTRODUCTION, DISCUS is an interpretive man-computer interface system, currently implemented as a conversational CAI language. It is programmed entirely in assembly language, for the IBM 360 series. It is characterized by fast execution, economy of core, and ready interface with CRT-oriented time-sharing systems.

ARCHITECTURE See block diagram, page 115.

REQUIREMENTS

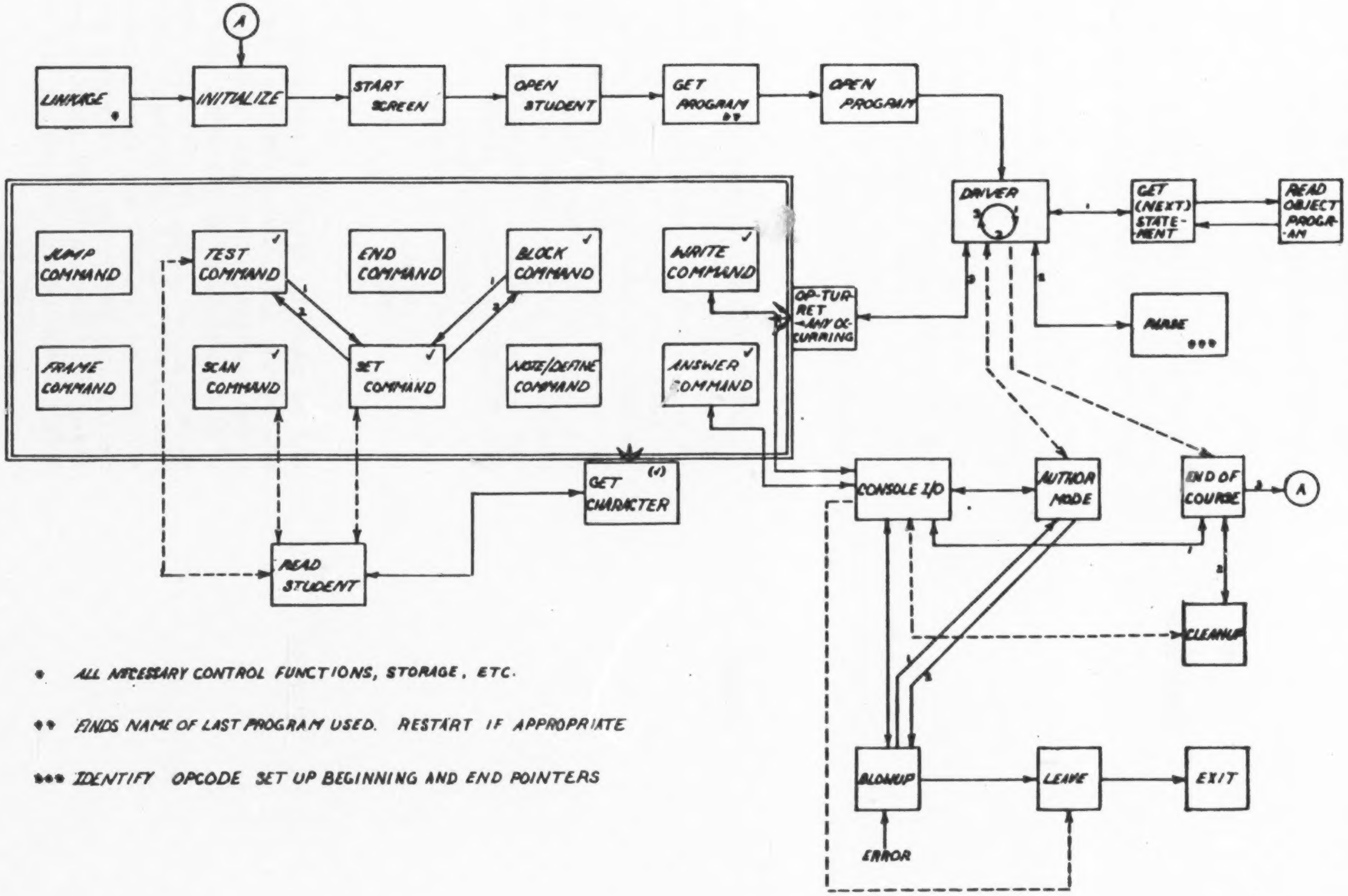
Source	80-byte card images in DISCUS source language.
Compiler	The program which converts DISCUS source code to DISCUS OBJECT code. Consists of approximately 8500 bytes of basic assembly language 2000 " for output buffers 800 " for each input buffer 26 " for each unique label (symbol) compiled 2n bytes for print buffers where n is the block-size of the SYSPRINT data set (see page 134, control card number 4).
Executor	The program which interprets the DISCUS OBJECT data set. Consists of approximately 8500 bytes of executable code 4900 " for each individual using the system.

(REQUIREMENTS, cont.)

Object Data Set	The data set into which the DISCUS OBJECT text is placed by the compiler. The format of the object data set is DCB=(RECFM=VB,BLKSIZE=1000)
Terminal	CRT displays which may be erased; written from top left corner to bottom right corner, with a variable number of characters; read, from either the top left corner to the last data byte on the screen, or from the first position of manually-entered data to the last position of manually-entered data. Any line from 2 to 100 bytes in length and a total screen of up to 1100 bytes can be accomodated.

CURRENT IMPLEMENTATION

DISCUS is currently implemented in time-sharing systems operating under IBM OS/360 (Release 17) at the Berkeley and Los Angeles campuses of the University of California. Terminal equipment in use at these locations, respectively, consists of a Sanders 730 CRT System and a CCI 30 CRT System.


A
LINKAGE *
INITIALIZE
START SCREEN
OPEN STUDENT
GET PROGRAM **
OPEN PROGRAM
DRIVER
1
2
3
GET (NEXT) STATE-MENT
READ OBJECT PROGR-AM
PARSE ***
OP-TURRET ANY OCCURRING
JUMP COMMAND
TEST COMMAND
END COMMAND
BLOCK COMMAND
WRITE COMMAND
FRAME COMMAND
SCAN COMMAND
SET COMMAND
NOTE/DEFINE COMMAND
ANSWER COMMAND
GET CHARACTER (1)
READ STUDENT
CONSOLE I/O
AUTHOR MODE
END OF COURSE
A
CLEANUP
BLOWUP
LEAVE
EXIT
ERROR

* ALL NECESSARY CONTROL FUNCTIONS, STORAGE, ETC.

** FINDS NAME OF LAST PROGRAM USED. RESTART IF APPROPRIATE

*** IDENTIFY OPCODE SET UP BEGINNING AND END POINTERS

GLOSSARY	The following definitions correspond with standard meanings for the terms defined, qualified to apply to DISCUS.
Statement	A statement is the smallest cohesive unit with which the DISCUS compiler will deal. The DISCUS COMPILER accepts statements, each of which must contain one and only one opcode (q.v.) and each of which must be terminated by a semicolon as an end-of-statement delimiter. A statement usually contains an operand (q.v.) between the opcode and the end-of-statement delimiter. A statement may be identified by a label (q.v.) which must itself be followed by a colon. Thus the format of a DISCUS statement is typically: LABEL: OPCODE OPERAND;
Opcode	A DISCUS OPCODE defines the nature of the operation which is to take place. If the statement includes an operand, at least one blank must be interposed between the opcode and the operand. If no operand is encoded, the opcode is followed by the statement-delimiting semicolon, either immediately or with one or more blanks interposed. Thus the following are all legal: OPCODE OPERAND; OPCODE OPERAND; OPCODE; OPCODE ;
Operand	Any of several types of parameter. These may be absolute data, symbolic labels, or codes peculiar to special operations. The OPCODE generally performs an operation either on or using an OPERAND. One or more blanks may be interposed between the end of the OPERAND and the statement-delimiting semicolon. (For the effect of doing this with a WRITE opcode, see page 22ff) Both of the following are legal: OPERAND; OPERAND ; The OPERAND must always be preceded by at least one blank.

Suboperand	This is a term peculiar to DISCUS, denoting portions of a SCAN operand which are logically separated by an ampersand (&), "or-bar" (|), or comma. Thus in the following: SCAN BEANS PEAS, CARROTS & CORN; the following constitute suboperands: BEANS PEAS , CARROTS and CORN.
Label	A LABEL is a character string used to identify and locate a statement. In the current implementation of DISCUS, it may not exceed 8 alphanumeric characters, of which the first must be alphabetical. It must end with a colon, or one or more blanks followed by a colon. It must not be broken by blanks. No more than one label may be attached to a single statement, nor may the same label be attached to more than one statement.
Word	A WORD is a string of characters which does not include imbedded blanks, special characters, or symbols, and which is surrounded by blanks, either explicit or implicit. WORDS used in SCAN statements constitute elements against which a user's response may be compared.
Literal	A LITERAL is a string of characters, punctuation marks, symbols, and explicit blanks, not used in a special code sense. In order to be treated as a LITERAL such a string must be surrounded by single quotation marks. (For use of these marks themselves as LITERALS, see example, page 26. Inclusion of a character in a LITERAL suppresses any special characteristics which it may normally possess in the DISCUS system.
Character String Variable	A CHARACTER STRING VARIABLE is a string whose length or content is variable and which is accessed by a label. Its contents may be changed during execution.

	The space for a character string variable must be pre-established by a separate defining statement.
Numeric Variable	A NUMERIC VARIABLE is a field intended to contain a numerical quantity whose magnitude may be changed during execution. The space for a numerical variable must be pre-established by a defining statement. The contents of a variable are accessed by referring to its label.
Match Block	A block of code which is entered only if a SCAN or TEST immediately preceding it has been satisfied.
Fail Block	A block of code which is entered only if a SCAN or TEST immediately preceding it has *not* been satisfied.
*Unconditional Block**	A block of code which is entered regardless of match or fail condition resulting from a SCAN or TEST. Entry into an UNCONDITIONAL BLOCK serves to increment the existing block level.
Block Level	A numerical value (1 to 250) assigned to individual nested blocks. Thus a block within a block which is within a block at level 1 will have a block level of 3.

---

*Sometimes called the "Block block."

OPCODES — The functions of the DISCUS OPCODES and their modifiers are defined in the pages which follow. Technical notes are added in some cases. The order of presentation is the same as that used in PART III.

WRITE or W — The WRITE opcode (command) causes the screen to be written from the top, after erasing all previous display material. If the number of characters in the statement exceeds screen capacity, it will write to the end of the screen and then wait until the console-user presses the interrupt button (or other designated signal, such as "send page" on the Sanders system). This action is treated as a continuation of the WRITE command; the screen is erased; and the remainder of the statement is displayed.

End-of-line formatting is automically performed.

WRITE(NF) or W(NF) — This command writes without end-of-line formatting.

WRITE(ND) or W(ND) — This command writes without first erasing the screen. The characters to be displayed are laid down beyond the previously-written text. The format or no-format style of the preceding WRITE command is continued in the WRITE(ND).

ANSWER or A — The ANSWER OPCODE sets in train the following operations:

1. It causes a carat to be displayed at the beginning of the line below the currently-displayed WRITE text, to invite keyboard input. The input itself is displayed as typed, without disturbing the WRITE display, and without being considered by the computer until "send" is signalled by the student.

2. After "send," ANSWER takes whatever has been typed* and puts it in a character variable whose name is "ANSWER." (This is not an ordinary label address, but a reserved word, predefined in the system.) The ANSWER field can contain as many as 250 characters, any excess being truncated on the right. A record is also kept of the total number of characters present in the ANSWER field.

During a SCAN only, in addition to the characters and blanks actually input, ANSWER inserts a blank at the beginning of the field, and one at the end. Thus the character-count is always increased by two.

At the start of program execution, the ANSWER field contains unknown information (usually nothing). Thereafter it contains the data recorded there on the occasion of the last "send." Whenever a new "send" is signalled, the new input (which may also be nothing at all) entirely displaces what was there before.

In order to save the contents of the ANSWER field at a particular stage during execution, they must be transferred to a defined character variable (see SET, p. 57). Meanwhile, however, they can be scanned successively for various elements, may be quoted in WRITE displayed by imbedding the reserved word "ANSWER" (always in double quotes) in the WRITE operands, may be added-to, and may have selected material copied out of them through the use of "starred variables" coded in SCAN operands (see p. 111).

---

*None of the constraints applying to the coding of reserved characters (single quotes, double quotes, etc.) are imposed on student input. If the student inputs any of these, they are automatically treated as literals.

ANSWER(NF) or A(NF) This opcode functions in exactly the same way as ANSWER, except that instead of placing a carat and cursor at the beginning of the line below the last line of WRITE text, it eliminates the carat altogether and places the cursor at the end of the WRITE text, or at the end of any elements "supplied" to the screen by the ANSWER operand (see p. 39). The student may back the cursor across such elements in order to fill in a blank anywhere in the ANSWER operand. This is illustrated in the following example:

```
Statement: - WRITE PLEASE SUPPLY THE MISSING WORDS IN THE FOLLOW-
                   ING FRAGMENT OF SHAKESPEARE':'/ WHEN TO THE
                   SESSIONS OF SWEET SILENT _______/ I SUMMON UP;

             A(NF) _____________ OF THINGS PAST;
```

```
Display: -

                   WHEN TO THE SESSIONS OF
                   SWEET SILENT __________
                   I SUMMON UP ___________
                   OF THINGS PAST_
```

Confronted with this display, the student would be able to move the cursor back as shown below, overwrite the solid line, and have his input considered as part of the ANSWER field:

```
Display: -

                   WHEN TO THE SESSIONS OF
                   SWEET SILENT __________
                   I SUMMON UP ___________
                   OF THINGS PAST
```

He could move the cursor further back, i.e., into the block of WRITE text, but his input will not be seen by the ANSWER statement. Accordingly, the example represents a poor formulation. Correctly encoded, the ANSWER operand should have been written

```
Statement: -       A(NF) _______I SUMMON UP_______OF
                         THINGS PAST.;
```

SCAN or SC

SCAN searches the ANSWER field for elements specified in its own operand. These elements may consist of words, strings of words, single characters, numerals, punctuation marks (coded as literals) and blanks (also coded as literals). A string containing a combination of words, blanks, punctuation marks, etc., if it is to be sought in exactly the same form as that in which it appears in the SCAN operand, needs to be specified as a literal in its entirety by surrounding it with single quotes. If intervening words in the ANSWER field are acceptable, the string should not be specified as a literal.

Statement	SCAN SODIUM CALCIUM;
Result	The ANSWER field will be searched for two separate words, sodium and calcium, in that order,
Statement	SCAN 'SODIUM CALCIUM,';
Result	The ANSWER field will be searched for the two words in that order, separated by a single blank and followed by a comma.

Successful match of a SCAN specification against some part of the ANSWER field is reflected by the setting of a condition code to "match". Failure sets the condition code to "fail".

The SCAN operand can be divided into two or more sub-operands, each of which is compared (in the order of their specification) against the contents of the ANSWER field until a specified combination succeeds, at which time the condition code is set to "match", scanning ceases, and processing jumps to the next statement.

Suboperands are separated from each other by commas, or "OR"-bars (|), both of which act as "or" logical operators, or by ampersands, which serve as "and" logical operators.

Statement	SCAN SODIUM, CALCIUM;
Result	The ANSWER field will be searched first for SODIUM. If found, the condition code will be set to "match" and execution will jump to the next statement beyond the delimiting semicolon. If not found, the ANSWER field will be searched for CALCIUM.

Statement	SCAN SODIUM & CALCIUM;
Result	The ANSWER field will be searched first for SODIUM, then for CALCIUM. The "match" condition code will be set only when both are found. The order in which the two appear in the ANSWER field will not affect the result.

When it is desired that the ANSWER field will be searched for elements which will match the contents of some variable, the latter can be specified in the SCAN operator simply by referring to the variable's label address, in double quotes.

Statement	SCAN "CHEMICAL";
Result	If CHEMICAL is the label of a character variable containing - say - PHOSPHATE, then PHOSPHATE is the word which will be compared against the ANSWER field, as it were another literal. If CHEMICAL is an arithmetic variable, the numerals making up its contents are treated as characters, in a literal string. Combinations of letters, words, blanks, punctuation marks, etc., in a variable's operand are always treated as a single literal string when expanded into a SCAN operand in this way.
Statement	(Assume CHEMICAL contains PHOSPHATE, SULPHATE) SCAN "CHEMICAL";
Result	Equivalent to that of SCAN 'PHOSPHATE, SULPHATE';

In the current implementation of DISCUS there is no way of activating, in a SCAN operand, commas or ampersands fetched from a variable. Thus the contents of a variable must always be scanned for in their entirety, as a literal, rather than as a group of suboperands.

A third logical operator usable in a SCAN operand is the "not" sign (¬). It has the effect of negating all elements in the suboperand in which it appears.

Statement	SCAN ¬ CAT;
Result	Any ANSWER field which does not contain CAT will be matched.

Statement	SCAN CAT ¬ MOUSE; or SCAN ¬ CAT MOUSE;
Result	Any ANSWER field which contains neither CAT nor MOUSE will be matched.
Statement	SCAN CAT & ¬ MOUSE;
Result	An ANSWER field containing CAT but not MOUSE will be matched, since the ¬ does not apply to the cat suboperand.
Statement	SCAN CAT, ¬ MOUSE;
Result	The ANSWER field will be searched first for CAT. If successful, the operation will terminate without checking for ¬ MOUSE. If unsuccessful, the ANSWER field will be searched for ¬ MOUSE, and if no MOUSE is found a match condition will be set.
Statement	SCAN ¬ MOUSE, CAT;
Result	The ANSWER field will be searched first for MOUSE. If no MOUSE is found, a match condition will be set and execution will jump over the CAT suboperand. If a MOUSE is found, scanning will continue to the second suboperand.

By scanning for parts of words as literals it is possible to obtain matches against misspelled words, in many cases. For such purposes the literal for a word-beginning must include the starting blank, the literal for a word-ending must include the ending blank, and the literal for a possible word-middle should include no blanks at all.

Statement	SCAN ' DOD';
Result	A match will be obtained if the ANSWER field contains any word beginning 'DOD...'
Statement	SCAN 'DRON ';
Result	A match will be obtained if the ANSWER field contains any word ending with '...DRON'
Statement	SCAN 'CAH';
Result	A match will be obtained if the ANSWER field contains any word containing that combination.

All three will succeed with DODECAHEDRON, for example.

DEFINE or D — DEFINE statements are used to establish, without initializing, the variables to be used in a particular DISCUS program. In each case the DEFINE opcode must be qualified in such a way as to establish whether the variable is to be used to contain mathematical data or characters:

```
DEFINE(A)    or  D(A)     -     arithmetic
DEFINE(C)    or  D(C)     -     character
```

In addition, the maximum length in number of characters (up to 250) must be specified for D(C), in order for the compiler to reserve adequate space for planned content.

DEFINE statement are always labelled, otherwise the variables which had been created would not be accessible. Typical defining statements of each type would be:

```
ADDO:  D(A);
CHAR:  D(C) 100;
```

SET or S The SET statement is used to initialize, alter, or clear the contents of a variable. The SET operand always contains three elements: an objective variable (to the left of an equal sign), the equal sign itself, and the material with which the variable is to be equated. Names of variables specified in SET operands are always separately surrounded by double quotes. Let us assume "EINSTEIN" and "RELATIVE" are arithmetic variables.

Statement	SET "EINSTEIN" = "RELATIVE";
Result	Whatever the contents of RELATIVE may be now become the sole contents of EINSTEIN.
Statement	SET "EINSTEIN" = "EINSTEIN" + "RELATIVE";
Result	EINSTEIN now contains its prior content plus the contents of RELATIVE. RELATIVE is unaffected in both cases.

The kind of variable (i.e., arithmetic or character) which is the object of the operation, and which is always named immediately after the opcode, determines whether the operation will be arithmetical or character-manipulative.

Statement	SET "ADDO" = 2 + 2 ;
Result	If ADDO has been defined through a D(A), it will now contain 4.
Statement	SET "CHAT" = '2 + 2';
Result	If CHAT has been defined through a D(C), it will now contain the literal 2 + 2 (2b̸+b̸2).

Words or strings placed in character variables must be specified as literals, as above. Otherwise the operation will fail, or have unspecified results.

Statement	SET "CHAT" = 2 + 2;
Result	CHAT is set to null.

If an attempt is made to place a character variable in an arithmetic variable, numerals and operators will be dealt with arithmetically, but all other characters will be converted to zero and disregarded.

Statement

```
SET "ADDO" = '20 QUESTIONS';
```

Result

ADDO will contain the number 20.

The presence or absence of blanks outside of integers and literals in any SET operand is immaterial.

Statements

```
SET "ADDO" = 128786+4;
SET "ADDO" =    128786    +    4    ;
```

Result

The same.

Statements

```
SET "CHAT" = 'A' 'B' 'C'     'ƀ'     ',D';
SET "CHAT" = 'ABCƀ,D';
```

Result

The same.

The contents of a variable may be altered directly, as in

```
SET "ADDO" = 4;
SET "CHAT" = 'HELLO, THERE!';
```

or indirectly, as in

```
SET "ADDO" = "MATH";
SET "CHAT" = "VERBOSE";
```

A character object variable is always set to the concatinated result of the expanded variables and the literals on the right side of the equal sign. Arithmetic variables are expanded and converted to character string equivalents before processing.

One or more arithmetic operations may be performed on the contents of an arithmetic variable, using the following operators:

+	add
-	subtract
*	multiply
/	divide
▒	anything else is considered a plus sign.

Arithmetic expressions are evaluated from left to right, with no hierarchy of operations being observed. All processing is in integer arithmetic, and all intermediate fractional results are dropped.

Statement

```
SET "ADDO" = "MATH"  + 10 / 13 + 2 * 5;
```

Result

(Assume "MATH" contains 7)
ADDO will be set at 15.

If the prior contents of ADDO are brought into the operation, the sequence of doing so is highly important. The result of

S "ADDO" = "ADDO" + "MATH" + 10 / 13 + 2 * 5 is quite different from the result of S "ADDO" = "MATH" + 10 / 13 + 2 * 5 + "ADDO".

In dealing with character variables, arithmetic operators have no effect:

Statements	SET "CHAT" = "VERBOSE" + "PROLIX"; SET "CHAT" = "VERBOSE""PROLIX";
Result	The same. The contents of VERBOSE are placed in CHAT, followed by the contents of PROLIX. (Neither VERBOSE nor PROLIX is affected)

In order to prevent concatination of elements packed into character variables, it is often necessary to specify blanks:

Statement	SET "LIST" = 'COFFEE' + 'TEA' + 'ROOT BEER';
Result	LIST will contain COFFEETEAROOT BEER.
Statements	SET "LIST" = 'COFFEE' 'ƀ' + 'TEA' + 'ƀ' + 'ROOT BEER'; or SET "LIST" = 'COFFEEƀ' 'TEAƀ' 'ROOT BEER';
Result	LIST will contain COFFEE TEA ROOT BEER.

Both arithmetic and character variables are cleared by omitting to specify anything to the right of the equal sign.

Statement	SET "ADDO" = ; or SET "ADDO" = 0;
Result	ADDO will contain zero (0).
Statement	SET "CHAT" = ;
Result	CHAT will contain nothing.

While numerical quantities may be subtracted from the contents of arithmetic variables, there is no way of subtracting characters from character variables except by truncation, that is, by transferring the entire string to another variable which has been so defined that there will not be room enough for all the characters, and the excess will be dropped off. A character string may be scanned, however, for particular elements, after first being transferred to the ANSWER field for the purpose.

Statements

```
SET " ANSWER" = "CHAT";
SC COFFEE;
```

Result

If "coffee" appears anywhere as a separate word in the string, it will be detected and a match condition will be set.

TEST or T

The TEST statement compares the contents of an object variable (specified to the left of a logical operator in the operand) with whatever is specified to the right of the operator. TEST much resembles SCAN, except that it must always succeed in its entirety in order to set the condition code to positive.

The logical operators used in TEST are:

=	equals
>	is greater than
<	is less than
¬	is not (=, >, and/or <)

and they may be used singly or in any combination.

Example 1: `T "X" = 4;`

Example 2: `T "ALPHA" ='ƀNOƀ';`

Example 3: `T "Y" > "Z" + "N" - 2`

JUMP or J

The JUMP statement transfers control unconditionally to a statement located elsewhere in the program, as specified by that statement's label address, entered (not in quotes) as the operand of the JUMP statement.

Statement

```
JUMP SASPARIL;
```

Result

Processing breaks sequence and jumps to the statement whose label is SASPARIL.

MATCH or M — Permits entry into the block of code which follows it only if the last preceding SCAN or TEST has succeeded, i.e., if the current condition code is positive. As each such block is entered, the level of processing drops to the next lower level. This brings into play a new condition code which can be modified without affecting the conditions that enabled entry into that block. In the current implementation of DISCUS, each statement in the block is processed but not executed until an END statement (q.v.) is encountered at the same level at which the "ignore mode" was initiated. Blocks may be nested to a depth of 250.

A MATCH statement may be suppressed after N executions by specifying N. The N-counter (hereinafter called the "MFB-counter") is reset by FRAME (q.v.).

FAIL or F — Permits entry into the block of code which follows it only if the current condition code is negative. Otherwise, FAIL operates in exactly the same way as MATCH.

BLOCK or B — Permits entry into the block of code which follows it, regardless of current condition code setting. BLOCK serves to drop the level of processing one level. Looping may be limited in the same way as is done with MATCH and FAIL, by an "MFB-counter."

FRAME or FR

The FRAME statement marks the beginning of a logical division of DISCUS code, and is usually labelled. It has no operand, and performs no overt processing. It does, however, reset all MATCH, FAIL, and BLOCK recursion counters ("MFB-counters") to zero, serves as a point of reference for USE statements accessing it, and serves as a restart location.

A "frame" as referred to in this manual comprises all the coding beginning with a FRAME statement and ending at the next FRAME statement in sequence. Normally a frame contains

a. Instructional test (W)
b. A question of requirement (W or W(ND))
c. Opportunity for student input (A)
d. Program reactions (SC, T, S, M, B, J, U, S, E, W, etc.)

In order for it to be meaningful as a conversational unit, it must contain at least (c) and some form of (d).

NOTE or N

The NOTE opcode causes its operand to be printed in the OBJECT listing. It is otherwise inert. In addition to preserving and drawing attention to programmers' memoranda, NOTE can be used as a temporary replacement for other opcodes, when it is desired to deactivate a statement temporarily without actually removing it from the source or from the object module. This can be done very simply by encoding Nƀ ahead of the existing opcode.

END or E

This opcode terminates a block which was entered at its own level (through a MATCH, FAIL, or BLOCK statement). To emerge from a nested block structure, a separate END statement is required for each upward step, and the total number of END statements must equal, but not exceed, the sum of all MATCH, FAIL, and BLOCK statements associated with that structure.

USE or U

The USE statement transfers control temporarily to one or more statements accessible through a label address specified in its operand. The scope of this instruction depends on the nature of the first statement encountered at the location addressed:

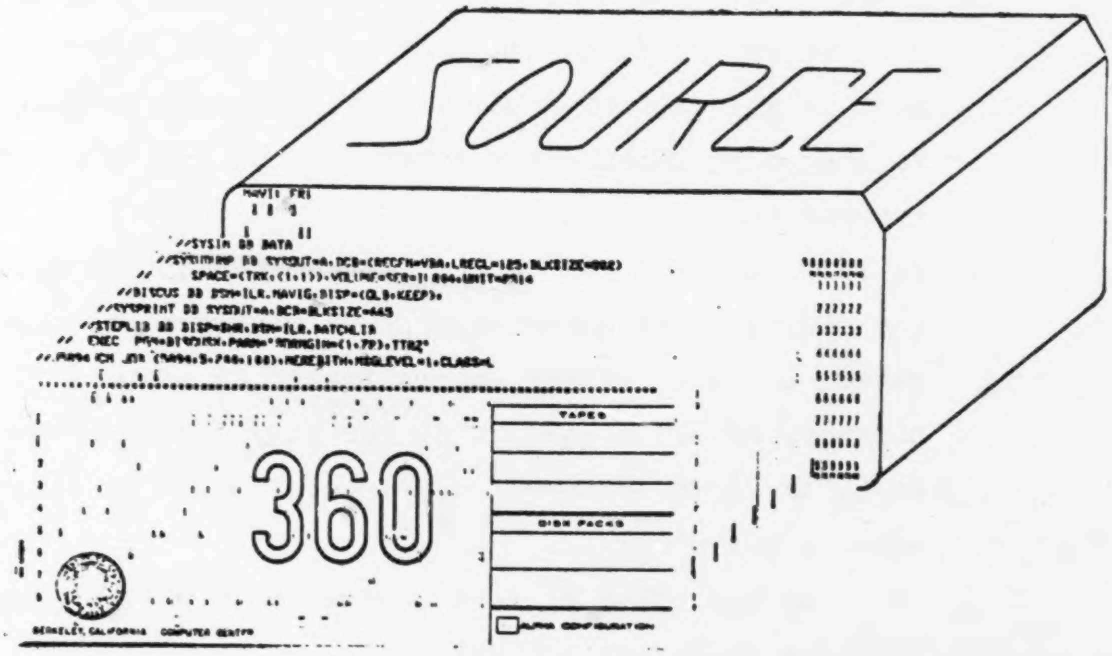
a. If it is a FRAME statement (q.v.) the entire frame will be executed, and as soon as the next FRAME statement is encountered processing will return automatically to the point immediately after the USE statement which invoked it.

b. If it is a MATCH, FAIL, or BLOCK statement, it will cause the entire block governed by that statement to be executed, and will return automatically as soon as it encounters an END statement at the same level as the MATCH, FAIL, or BLOCK statement by means of which it first entered the block.

c. If it is any other kind of statement it will process that one statement and return.

JOB CONTROL LANGUAGE FOR COMPILING DISCUS

In order to compile a DISCUS source program in either the UC (Berkeley) or UCLA systems, certain Job Control statements must precede and follow the source deck when it is read into the IBM/360.


SOURCE
//SYSIN DD DATA
//STEPLIB DD DISP=SHR,DSN=ILR.BATCHLIB
360
TAPES
DISK PACKS
ALPHA CONFIGURATION
BERKELEY, CALIFORNIA COMPUTER CENTER

The following control cards apply to the system implemented at Berkeley:

1. //J5894JCM JOB (5894,5,200,100),MEREDITH,MSGLEVEL=1,CLASS=L

   A standard JOB card for IBM 360 running under OS/360 (version 17).

2. // EXEC PGM=DISCUS,PARM='SORMGIN=(1,72),TTRZ'

   An EXEC card specifying the normal production DISCUS compiler and PARMs for special functions (see Note #11 below).

3. //STEPLIB DD DISP=SHR,DSN=ILR.BATCHLIB

   A STEPLIB card defining the library in which the DISCUS compiler resides.

4. //SYSPRINT DD SYSOUT=A,DCB=BLKSIZE=n (n is any integral multiple of 133)

   A SYSPRINT card specifying where the listing of the program as compiled is to be written.

5.
```
//DISCUS DD DSN=ILR.NAVIG,DISP=(OLD,KEEP),
        SPACE=(TRK,(1,1)),VOLUME=SER=ILR04,UNIT=2314
```

The data set in which the compiled DISCUS object code is to be placed for later execution.

6.
```
//SYSUDUMP DD SYSOUT=A,DCB=(RECFM=VBA,LRECL=125,BLKSIZE=882)
```

For compiler or system error only.

7.
```
//SYSIN DD DATA
```

A SYSIN card indicating that the DISCUS source program follows.

8. Deck of source statements.

9.
```
/*
```

Marks end of DISCUS source.

10.
```
//
```

Marks end of program.

11. The PARM field may contain any combination of the following, in any order separated by commas (no blanks):

    a. SORMGIN=(n,nn) Sets the margins of the source card images, "n" being the number of the column on the punch card for starting the image (as low as 1) and "nn" being the number of the column for ending the image (may be as high as 80). (If not specified, (1,80) is assumed.)

    b. TTRZ Generates a table of correspondence between statement numbers and the actual direct access locations (relative) of the statements as compiled. Both are printed alongside the statements in the object listing. The TTRZ number provides ready access to any statement through AUTHOR MODE (see PART V.)

    c. SNAP For system debugging only.

    d. STOP=stopcharacter When the single byte stopcharacter is encountered in a source card image, processing continues at the beginning of the next card image.

    e. DEBUG For system debugging only. A listing of all DISCUS statements, with special flags, is produced.

    f. SYMBOLS=number The absolute maximum number of symbols for which utility space will be reserved is here specified. If not specified, all the core storage in the program's region will be used for symbol table storage.

PART V - SYSTEM AUTHOR MODE

DESCRIPTION

System Author Mode, is a mode of operation available at the student terminal by means of which a DISCUS user (in the sense of an author/instructor, programmer, editor, etc.) can artificially influence execution and can arbitrarily jump from one part of the program to another. It may be invoked by entering the special command //A=Y at the terminal. This command may be entered at a carat or, if there is none, in the top left hand corner of the screen.

DISTINGUISHED FROM PROCTOR AUTHOR MODE

In Part III, as one of the subroutines useful in the programmer's repertoire, we suggested one which would permit a user to jump execution to another part of the program without using DISCUS System Author Mode. This, Proctor Author Mode, device is essentially custom-made for a particular instructional sequence, as it entails coding a SCAN command and a JUMP command for every unique location (according to label address) to which one wishes to be able to jump.

This has its advantages and disadvantages.

On the one hand, Proctor Author Mode is a convenient way of getting around in the program without worrying about the precise disc location of the object statements (the TTRZ number). If the user makes a mistake in designating a label, he is so informed, rather than being transferred to some wrong point in the program. The labels can be directly keyed to written text and to the author/instructor's outline, and of course appear on all object listings opposite the statements to which they are affixed, as well as in a separate alphabetical list showing the numbers of all statements referring to them.

On the other hand, in a program of any size, the work of encoding and punching the hundreds of SCAN and JUMP statements involved in a Proctor Author Mode routine is considerable, and the resulting code is good only for one specific set of labels. Moreover, it must be updated whenever the program itself is substantially changed. The processing of this routine is fairly expensive in terms of computer time: in the case of one of our CAI courses, having 297 labels accessible by this method, the IBM 360/40 takes up to 23 seconds of real time to get to the last label on the list.

But its most serious weakness is that it doesn't reveal what is going on inside the program, in the way the System Author Mode does. Proctor Author Mode merely transfers from normal execution in one place to normal execution in another.

DIAGNOSTIC DISPLAY

An integral part of SYSTEM AUTHOR MODE is the diagnostic display which appears on the screen whenever invoked through a System Author Mode command from the terminal, and also whenever an unexecutable statement occurs during processing. Examples of both are shown on page 140.

SYSTEM AUTHOR MODE COMMANDS

The following SYSTEM AUTHOR MODE commands are available to the user:

//A=Y

Enter author mode after execution of each statement.

Other author mode commands do not depend on //A=Y having first been entered, but when not under //A=Y processing will revert to normal processing at the first opportunity. After specifically invoking author mode through //A=Y, processing remains in author mode until revoked.

//A=N	Turns off author mode.
//J=N	Suppresses all JUMP statements that would normally be executed.
//J=Y	Reinstates the operation of JUMP statements.
//S=number	Causes processing to stop at the statement number specified (per OBJECT listing). May be pre-set at any time.
//V=block numberƀoffset	(works only under //A=Y) Causes the contents of a character variable to be displayed. The user must specify the block number and the offset in that block in order to retrieve the variable from the student data set. The information describing the location can be found in the source listing.
//F=TTƀRƀZ	This is the basic "jump" command in System Author Mode. The "TTRZ" refers to the disc location of each statement, which differs by one "R" from that indicated in that object listing. Thus if the user wants to go to the statement whose TTRZ as shown in the object listing is 1/6/30, he should input //F=1ƀ7ƀ30. The true location also differs by one "Z" from the TTRZ as shown in the diagnostic display. In case of doubt, use the number as shown in the object listing and advance to the desired location through successive interrupts.

EXIT (Berkeley)
END (UCLA)

Exits from the program. The student data set is preserved, and a notation of the restart location (last frame) is recorded.

EDITING

There currently is no dynamic editing facility within the DISCUS system. That is, one cannot change the OBJECT MODULE from the terminal unless such a facility is available <u>outside</u> DISCUS, i.e., in the time-sharing monitor which controls the overall operation. (UCLA's "URSA" monitor does provide this.)

DISCUS DIAGNOSTIC SCREEN

Typical
Display

```
REQUESTED AUTHOR MODE
OPERATION  STATM   COND   LEVEL   TTRZ+1
   S         7     FAIL     1     0-1-8

----------------------------------------
"ZAP" = 1
```

Interpretation:

The executor has just executed the 7th statement in the program which is a <u>SET</u> statement. The condition code for current execution purposes is FAIL. Execution is proceeding at level 1. The statement occupies disc storage position 0-1-7. Underneath the line of hyphens is displayed the operand of the statement in question.

PART VI EXERCISES

page

A. WRITE STATEMENTS AND ANSWER STATEMENTS 143

B. SCAN STATEMENTS 147

C. USE OF LOGICAL OPERATORS IN SCANS 151

D. VARIABLES 153

E. DECISIONS 160

HOW TO USE THIS CHAPTER

Space is provided (either as a display screen or as a few blank lines) for you to write your answer to each question as you proceed through this section.

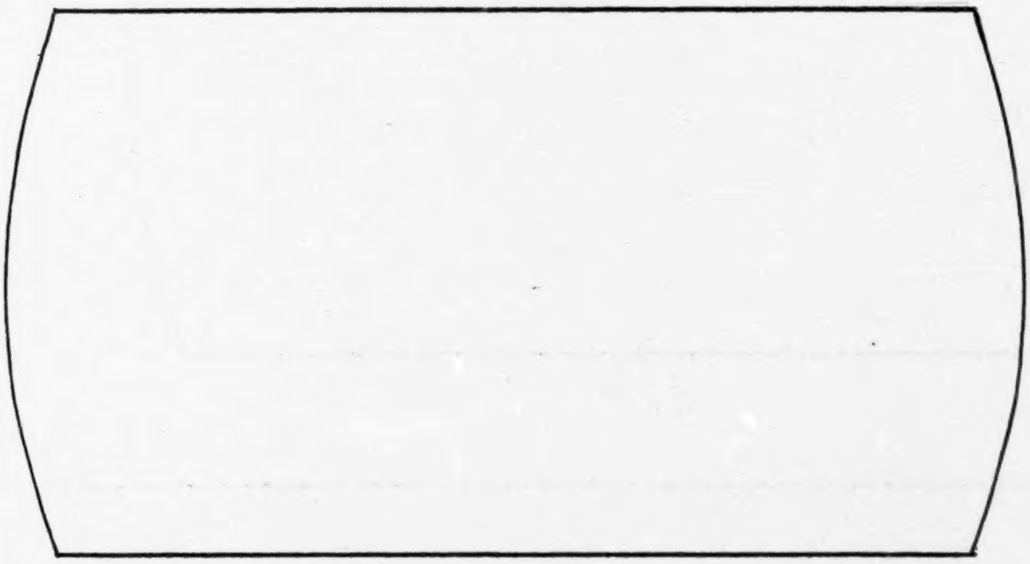
For the most part, solutions to problems are provided on the lower part of the same page, below a broken line. You should cover everything below the broken line until you are ready to check your answers.

A.

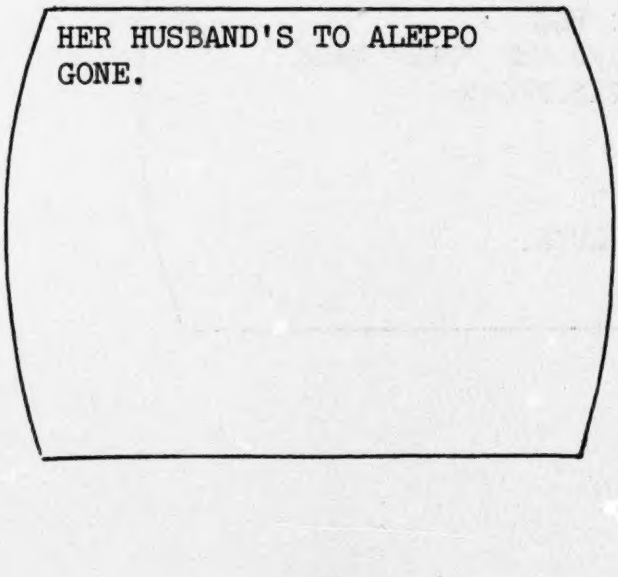
WRITE statements and ANSWER statements.

1. How will the following appear on the screen? (Assume a 27-character-per-line screen capacity.)

```
WRITE HER HUSBAND''S TO ALEPPO GONE.;
```



1.



```
HER HUSBAND'S TO ALEPPO
GONE.
```

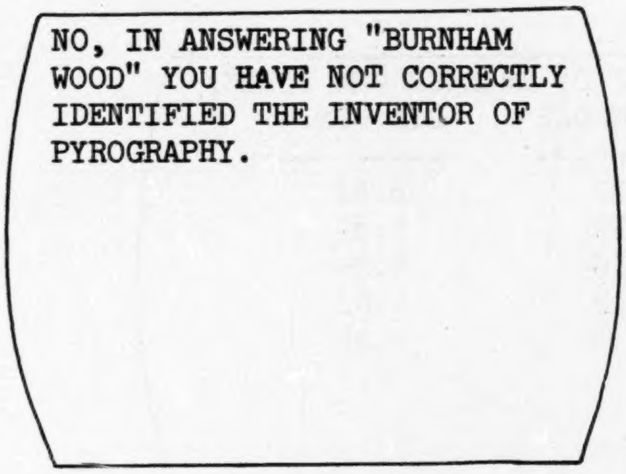
2. How will this appear? Show location of cursor after execution.

```
WRITE FOR '"MRS. CRUMLEY''S"' FRITTERS USE THE FOLLOWING
INGREDIENTS':'/ƀƀƀƀ1 EGG/ƀƀƀƀ1'/'2 CUP ''PET'' MILK/
ƀƀƀƀ2LB.FLOUR///GOOD LUCK.;
```

2.

```
FOR "MRS. CRUMLEY'S"
FRITTERS USE THE FOLLOWING
INGREDIENTS:
    1 EGG
    1/2 CUP 'PET' MILK
    2LB.FLOUR



GOOD LUCK.
```

3. How might the following display have been encoded?

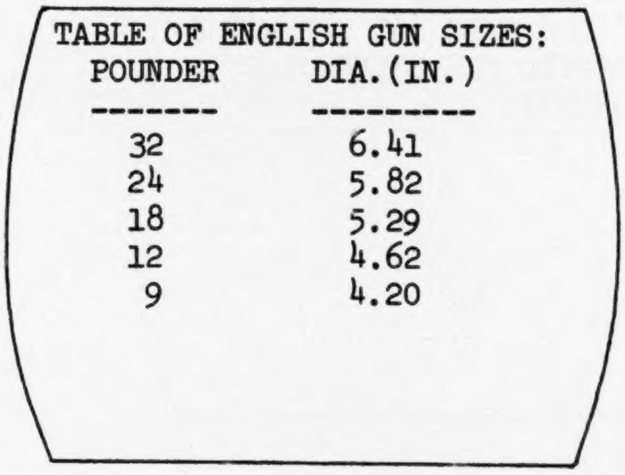


NO, IN ANSWERING "BURNHAM
WOOD" YOU HAVE NOT CORRECTLY
IDENTIFIED THE INVENTOR OF
PYROGRAPHY.

4. How would you add the words "TRY AGAIN" to the above screen?

```
3.  WRITE NO, IN ANSWERING '"'"ANSWER"'"' YOU HAVE NOT CORRECTLY
    IDENTIFIED THE INVENTOR OF PYROGRAPHY.;

4.  WRITE(ND) TRY AGAIN.;  or  W(ND) TRY AGAIN.;
```

5. How would you encode the statement creating the following display? (Use ƀ to indicate blanks.)



```
TABLE OF ENGLISH GUN SIZES:
  POUNDER     DIA.(IN.)
  -------     ---------
    32           6.41
    24           5.82
    18           5.29
    12           4.62
     9           4.20
```

____

____

6. What opcode might be useful for displaying long numbers of formulae, or to get the maximum number of words into a screen?

____

---

5. WRITE TABLE OF ENGLISH GUN SIZES ':'/ƀƀPOUNDERƀƀƀƀƀƀDIA.(IN.)/ƀƀ-------ƀƀƀƀƀ---------/ƀƀƀƀ32ƀƀƀƀƀƀƀƀƀƀƀ6.41/ƀƀƀƀ24ƀƀƀƀƀƀƀƀƀƀƀ5.82/ƀƀƀƀ18ƀƀƀƀƀƀƀƀƀƀƀ5.29/ƀƀƀƀ12ƀƀƀƀƀƀƀƀƀƀƀ4.62/ƀƀƀƀƀ9ƀƀƀƀƀƀƀƀƀƀƀ4.20/;

6. WRITE(NF); or W(NF);

B.

SCAN STATEMENTS

[A] Which of the following SCANs will intercept (i.e., match, or succeed with) the ANSWER field contents shown?

	SCAN statement	ANSWER field
1. M__F__	SC BARE CHOIRS;	BAREƀƀƀCHOIRS
2. M__F__	SC EXPOSTULATION REPLY;	REPLY AND EXPOSTULATION
3. M__F__	SC ' EXPO' 'ON ' REPLY;	EXPLOSION REPLY
4. M__F__	SC 'POSTU' REPLY;	YOUR EXPOSTULATION MERITS A REPLY....
5. M__F__	SC QUINQUIREME OF NINEVEH FROM DISTANT OPHIR;	QUINQUERIME OF NINEVEH FROM DISTANT OPHIR
6. M__F__	SC QUIT;	I'M TIRED AND HUNGRY AND I WANT TO QUIT!!!

(solutions to [A] 1-6)

1. Match. Intervening words and/or blanks do not affect the operation.

2. Fail. Elements must be in the stated order.

3. Fail. The literal - ' EXPO' is distinctive enough to screen out "explosion." (However, "exposition" would have matched.)

4. Match. The literal 'POSTU', taken from the middle of the word, is more distinctive than would be a literal at either of the ends. Neither "explosion" nor "exposition" would have got through.

5. Fail. Difficult input ruined by single spelling error. A more selective treatment would have been more practical, e.g., SC ' QUINQ' 'VEH ' DISTANT ' OPH';

6. Match. Preceding, intervening, or following words and/or blanks do not affect the operation, nor do the three exclamation marks.

([A] cont.)

7. M__F__	SC QUIT;	I'M NOT QUITE SURE
8. M__F__	SC T;	THE STATEMENT IS TRUE
9. M__F__	SC ' T';	THE STATEMENT IS TRUE
10.M__F__	SC 'T ';	THE STATEMENT IS TRUE

(solutions to [A] 7-10)

7. Fail. The SCAN demands a blank after the "t" of "QUIT". To match both QUIT and QUITE the element should be specified as a literal: 'QUIT'.

8. Fail. Same problem. The SCAN demands a blank on both sides of the "t".

9. Match. Will match any answer containing a word beginning with T. In this case, it caught on TRUE.

10.Match. Will match any answer containing a word ending with T. In this case, it caught on STATEMENT.

[B] After displaying the question "Who was Horace Greeley?", suppose you want to intercept answers of the following sense: - "He was a newspaper publisher." Formulate a SCAN operand which would best suit this purpose. (Disregard possibility of negatives.)

SC ____;

[C] Formulate a SCAN similar to the above, but with the added feature that it will succeed only if the answerer also mentions the (New York) Tribune.

SC ____;

[D] Formulate a SCAN which will ensure that the student uses a semi-colon somewhere in his reply.

____;

(solutions to [B] through [D])

[B] SC ' NEWSP';

[C] SC ' NEWSP' & ' TRIB';

[D] SC ';';

[E] Formulate a SCAN which will ensure the student's absolute accuracy in inputting the following:

It's not the 'eavy 'auls that 'urts the 'osses 'ooves; it's the 'ammer, 'ammer, 'ammer of the 'ard, 'ard 'ighways.

SC ____

____

(solution to [E])

[E]
```
SC 'IT''S NOT THE ''EAVY ''AULS THAT ''URTS THE ''OSSES ''OOVES;
IT''S THE ''AMMER, ''AMMER, ''AMMER OF THE ''ARD, ''ARD ''IGHWAYS.';
```

(Explanation: The lone single quotes at the beginning and end of the string define the string as a literal, so that the semicolon and commas need not be separately enclosed in single quotes. This does not suffice for the apostrophes, however, because they are always regarded as on-off switches for literal, whenever they occur. The basic rule applies, that a single quote, in order to be treated as a literal must be doubled, whether or not it is inside a literal string.)

C.

USE OF LOGICAL OPERATORS IN SCANS

[A] Which of the following would succeed? Which fail?

	SCAN statement	ANSWER field
1. M__F__	SC FISH, FOWL, GOOD RED HERRING;	HAMILTON FISH
2. M__F__	SC FISH, FOWL, GOOD RED HERRING;	FOWL AND FISH
3. M__F__	SC FISH, FOWL, GOOD RED HERRING;	CERTAINLY NOT HERRING!
4. M__F__	SC DIAMONDS & EMERALDS & AMETHYSTS;	A CARGO OF DIAMONDS AND EMERALDS
5. M__F__	SC DIAMONDS & EMERALDS & AMETHYSTS;	DIAMONDS AND AMETHYSTS BUT ABSOLUTELY NO EMERALDS OR TOPAZES
6. M__F__	SC SANDALWOOD & CEDARWOOD, AND SWEET WHITE WINE	SANDALWOOD AND SWEET WHITE WINE

(solutions to [A] 1-6)

1. Match. Succeeds on FISH.

2. Match. Succeeds on FISH. Skips FOWL and GOOD RED HERRING suboperands.

3. Fail. Only part of the third suboperand is present.

4. Fail. AMETHYSTS is missing in the "and"ed suboperand.

5. Match. All three "and"ed suboperands are present. Permuted order is acceptable, because ampersands are used in the SCAN statement.

6. Match. Fails on the first "and"ed pair of suboperands, because CEDARWOOD is not present. Succeeds on third suboperand because AND SWEET WHITE WINE is present. Note that the coder may actually have been thinking of the commas as a comma rather than as an "or", and failed to turn it into a literal with single quotes.

(Use of logical operators in SCANS, cont.)

7. M__F__	SC SANDALWOOD & CEDARWOOD AND SWWET WHITE WINE	CEDARWOOD BUT NO SANDALWOOD
8. M__F__	SC ¬NOT, ILLYRIA;	THIS IS ILLYRIA, LADY
9. M__F__	SC ARCADIA, ¬NOT;	THIS IS ILLYRIA, LADY
10.M__F__	SC ILLYRIA, ¬ILLYRIA;	THIS IS ILLYRIA, LADY
11.M__F__	SC ¬NOT & BREAD;	MAN DOES NOT LIVE BY BREAD ALONE
12.M__F__	SC ¬EAT & BREAD;	MAN DOES NOT LIVE BY BREAD ALONE
13.M__F__	SC BREAD & ¬EAT, NOT;	MAN DOES NOT EAT BREAD ALONE

(solutions to [A] 7-14)

7. Match.	Succeeds on the "and"ed pair of suboperands. Inversion is acceptable because of the ampersand.
8. Match.	NOT is not present, so the statement succeeds on the first suboperand.
9. Match.	Fails on first suboperand, goes on and succeeds on second.
10.Match.	Succeeds on first suboperand, skips the second.
11.Fail.	NOT is present, so the first of the two "and"ed suboperands fails, therefore all fails.
12.Match.	EAT is not present; BREAD is.
13.Match.	EAT is present, therefore the two "and"ed suboperands fail. NOT is present, so the third suboperand obtains a match.

D.

VARIABLES

For the exercises in this section, all character variables will be labelled CHARA(number): and all arithmetic variables will be labelled ADDOX(number):.

1. Define CHARA0 in such a way that it has room for 95 characters.

____

2. Define ADDOX0 in such a way that it has room for 100,000.

____

3. Encode a statement placing the following character string into CHARA0:

THE CURRENT ENROLLMENT AT UCLA ISb

____

4. Encode a statement placing the number 25,000 into ADDOX0.

____

(solutions to [D] 1-4)

1. CHARA0: D(C) 95;
2. ADDOX0: D(A); (There is no way to specify the maximum size of an arithmetic variable)
3. SET "CHARA0" = 'THE CURRENT ENROLLMENT AT UCLA ISb';
4. SET "ADDOX0" = 25000; (Don't use commas)

(variables, cont.)

5. Encode a statement that will concatenate the number in ADDOX0 to CHARA0.

6. How would you display to the student the contents of CHARA0 at this point?

7. What would appear on the screen?

8. Another student signs up. How do you increment ADDOX0?

9. How do you bring CHARA0 up to date? (Use additional variables as needed.)

(solutions to [D] (5-9)

5. SET "CHARA0" = "CHARA0" "ADDOX0";

6. WRITE "CHARA0";

7. THE CURRENT ENROLLMENT AT UCLA IS 25000;

8. SET "ADDOX0" = "ADDOX0" + 1;

9. CHARA1: D(C) 34;
   SET "CHARA1" = "CHARA0"; (Only the first 34 character of CHARA0 are stored in CHARA1);
   SET "CHARA0" = "CHARA1" "ADDOX0";

(Variables, cont.)

10. As part of the sign-in procedure, you might ask the student to type his last name, following which you ask him to type his first name and initial. How would you store this information in both straight and inverted forms?

    (Step 1 - storing first ANSWER field)

    (Step 2 - storing second ANSWER field)

    (assume character variables as necessary have been defined)

11. You decide to create a separate list of students' last names, and, in order to get as many as possible into a single variable, to save only the first seven characters of each. How might you arrange this? (Feel free to define additional variables as needed.)

(solutions to [D] 10 and 11)

10. (Step 1)

```
SET "CHARA2" = "ANSWER";
```

(Step 2)

```
SET "CHARA3" = "ANSWER"  ' ,'  "CHARA2";

SET "CHARA2" = "CHARA2"  ' '  "ANSWER";
```

11.

```
CHARA4: D(C) 7;

CHARA5: D(C) 255;

          S "CHARA4" = "ANSWER";

          S "CHARA5" = "CHARA5"  ' '  "CHARA4";
```

(Variables, cont.)

12. If all the names are at least seven characters long, how many can be stuffed into the variable, assuming one blank is left between each seven character name?

13. How can you find out if a new student has a name identical with someone already on the list?

    To begin:

    ```
    W WRITE YOUR LAST NAME, PLEASE.;

    A;
    ```

14. Place the number 144 into ADDOX2, and the number 37 into ADDOX3.

(solutions to [D] 12-14)

12. 31

    255 / (1+7)

13.
    ```
    W TYPE YOUR LAST NAME, PLEASE.;

    A;

    S "CHARA4" = "ANSWER";

    S "ANSWER" = "CHARA5";

    SC "CHARA4";
    ```

    MATCH will occur here if the new name is identical to one of the old names.

14.
    ```
    S "ADDOX2" = 144

    S "ADDOX3" = 37;
    ```

(Variables, cont.)

15. Place the mean of ADDOX2 and ADDOX3 into ADDOX4.

____

ADDOX4 will contain ____________________________

(solution to [D] 15)

15. S "ADDOX4" = "ADDOX2" + "ADDOX3" / 2;

ADDOX4 will contain 90. (144 + 37/2 [note truncation])

(variables, cont.)

16. Suppose you want the student to construct and solve a problem in cubic measurement, which the program would check for accuracy. Five inputs are involved. Fill in the blank items. (Solution on next page.)

```
W WHAT LINEAR MEASURE WILL YOU USE?;

A;

S ________________________________;

W HOW WIDE IS THE OBJECT YOU HAVE MEASURED?;

A;

S ________________________________;

W HOW LONG IS IT?;

A;

S ________________________________;

W HOW HIGH IS IT?;

A;

S ________________________________;

S ________________________________;

W HOW MANY CUBIC________DO YOU GET?;

A;

S ________________________________;

SC "ADDOX__";
```

(Variables, cont.)

(solution to [D] 16)

```
16.  W WHAT LINEAR MEASURE WILL YOU USE;
     A;
     S "CHARA6" = "ANSWER";
     W HOW WIDE IS THE OBJECT YOU HAVE MEASURED?;
     A;
     S "ADDOX5" = "ANSWER";
     W HOW LONG IS IT?;
     A;
     S "ADDOX6" = "ANSWER";
     W HOW HIGH IS IT?;
     A;
     S "ADDOX7" = "ANSWER";
     S "ADDOX8" = "ADDOX5" * "ADDOX6" * "ADDOX7";
     W HOW MANY CUBIC "CHARA6" DO YOU GET?
     A;
     SC "ADDOX8";
```

(If a match is obtained with this SCAN, the following statement might be executed:)

```
     W CORRECT.;
```

If not, the following:

```
     W THAT''S ODD.  I GET "ADDOX8" CUBIC "CHARA6".;
```

E.

DECISIONS

The second group of exercises dealt with one form of decision making statement: the SCAN statement.

The other decision making statement is the TEST statement. TEST is used for comparing all of the contents of one variable with all of the contents of another. It can be used for the comparison of both character and arithmetic variables.

The following DISCUS rules are worth reviewing before starting the exercises in this group:

1. TESTS and SCANS may be performed at any point in the program.
2. The two items to be compared in a TEST are always explicitly stated.
3. A SCAN always compares its operand with the current ANSWER field.
4. The result of a TEST or of a SCAN is always match or fail, (success or non-success).
5. This result sets a condition code in the computer, which is tested by subsequent MATCH or FAIL statements.
6. MATCH and FAIL statements concern themselves only with the preceding TEST (or SCAN) on the same level.
7. A MATCH statement at a given level passes execution to the next statement if (and only if) the condition code indicates success. Otherwise it passes control to the statement following the next END statement on its own level.
8. A FAIL statement is precisely the same as a MATCH statement except that execution proceeds to the next sequential statement if the condition code indicates fail.

Problems:

1. What would the following routine decide: What action(s) would ensue?

```
W WHAT FITTINGS ARE USUALLY ASSOCIATED WITH PINTLESS?;

A;

SC GUDGEONS;

M;

W CORRECT;

S "NAUTICA" = "NAUTICA" + 1;     (assume NAUTICA previously defined
                                 as an arithmetic variable)

E;

F;

W I''M AFRAID YOU DIDN''T STUDY CHAPTER 4.;

E;
```

---

1. The routine would decide whether the student has used the word "gudgeons" in his reply. Action, if so, would be to display "Correct" and to increment an arithmetic variable which had been defined and labelled NAUTICA. Action, if not so, would be to display the words "I'm agraid you didn't study chapter 4."

(Decisions, cont.)

2. What would the following routine decide, and what action(s) would ensue?

```
         W THE THREE MAJOR TYPES OF FILM USED IN MICROGRAPHY ARE

           ____________________________________________________;

HAWKEN:  A;

         SC SILVER & DIAZO & VESCICULAR;

         M;

         W GOOD. YOU REMEMBERED ALL THREE.;

         JUMP BAR;

         E;

         SC SILVER & DIAZO, SILVER & VESCICULAR, DIAZO & VESCICULAR;

         M;

         W SILVER HALIDE, DIAZO, AND VESCICULAR.;

         J BAR;

         E;

         SC SILVER;

         M;

         W SILVER HALIDE IS RIGHT. THE OTHER TWO ARE DIAZO AND VESCICULAR.;

         J BAR;

         E;

         SC DIAZO;

         M;

         W DIAZO IS RIGHT. THE OTHER TWO ARE SILVER HALIDE AND VESCICULAR.;

         J BAR;

         E;

         SC VESCICULAR;

         M;
```

```
     W VESCICULAR IS RIGHT. THE OTHER TWO ARE SILVER HALIDE AND DIAZO.;

     J BAR;

     E;

     F;

     W ONE OF THEM USES A PHOTOSENSITIVE METALLIC SALT EMULSION/
       ONE OF THEM USES A PHOTOSENSITIVE DYE/
       ONE OF THEM USES BUBBLES /// TRY AGAIN.;

     J HAWKEN;

     E;

BAR: W WE WILL NOW SING HYMN NUMBER 35;
```

The form of "verbal flowchart" furnished on the page which follows is useful for this kind of exercise.

FRAME LABEL:__________ WORKSHEET

Test: Then:

( ) Whether

______________________________ If yes, ______________________________

____

If not, ______________________________

( ) Whether

______________________________ If yes, ______________________________

____

If not, ______________________________

( ) Whether

______________________________ If yes, ______________________________

____

If not, ______________________________

( ) Whether

______________________________ If yes, ______________________________

____

If not, ______________________________

( ) Whether

______________________________ If yes, ______________________________

____

If not, ______________________________

( ) Whether

______________________________ If yes, ______________________________

____

If not, ______________________________

( ) Whether

______________________________ If yes, ______________________________

____

If not, ______________________________

(Decisions, cont.)

(solution to [E] 2)

FRAME LABEL:_______ WORKSHEET

Test:

(1) Whether

student inputs all 3 If yes, Display "Good, etc.";

jump to BAR

If not, Go to next scan (2)

(2) Whether

student inputs any 2 If yes, Display complete answer;

jump to BAR

If not, Go to next scan (3)

(3) Whether

student inputs "silver" only If yes, "Silver is right...etc.".

jump to BAR

If not, (4)

(4) Whether

student inputs "diazo" only If yes, "Diazo is right...etc.".

jump to BAR

If not, (5)

(5) Whether

student inputs "vescicular" only If yes, "Vescicular is right...etc.".

jump to BAR

If not, (6)

(6) Whether

student input was unrecognized If yes, Display hint; jump back to

location of ANSWER statement.

If not, (not applicable in this case.)

( ) Whether

_______________ If yes, _______________

____

If not, _______________

The next three problems deal with this same example.

3. Bracket the MATCH and FAIL blocks in the coded "source" program, as in this example:

```
┌─ M;
│  W GOOD. YOU REMEMBERED ALL THREE.;
│  J BAR;
└─ E;
```

Does the number of END opcodes equal the sum of MATCH and FAIL opcodes?

Assuming that the sequence begins at condition code level 1, indicate the condition code level applying to each block.

4. Was the last block - the FAIL block - really necessary as such?

(solutions to [E] 3-4)

3. The number of END opcodes (6) equals the sum of MATCH (5) and FAIL (1) opcodes.

   Condition level 2 is in effect in all blocks, since none of them are nested.

4. No. The last entry on your decision worksheet should indicate this. The WRITE statement and the JUMP back to the ANSWER statement will invariably operate whenever execution reaches them. Execution can reach them only if none of the preceding MATCH blocks has been successfully entered.

   FAIL block should be used principally to identify a class of failure, rather than total failure. This implies that one or more general "cleanup" statements is often needed before proceeding to the next question.

5. How might you improve the scan for "vescicular", assuming that accurate spelling is not a prime requisite here?

(solution to [E] 5)

5. Probably "vescicular" would be misspelled oftenest as

vesicular

vesiculler

vesiculer

vesciculler

vescicullar

vesciculer

So the most vulnerable letters are those underlined below:

VESCICULAR

These can be "forgiven" by encoding the word as 3 literal strings or required elements

' VES''ICU''R '

(Decisions, cont.)

Note: the sequence of code of question 2 (page 162) may be written so that the scanning and writing functions are separated, as follows:

```
        W THE THREE MAJOR TYPES OF FILM USED IN MICROGRAPHY ARE

          _____________________________________________________;

        S "EMU" = 0;

HAWKEN: A;

        SC SILVER, DIAZO, VESCICULAR;

  1     M;

        SC SILVER;

  2     M;

        S "EMU" = "EMU" + 1;

        E;

        SC DIAZO;

  3     M;

        S "EMU" = "EMU" + 2;

        E;

        SC VESCICULAR;

  4     M;

        S "EMU" = "EMU" + 4;

        E;

        T "EMU" = 7;

  5     M;

        W GOOD, YOU REMEMBERED ALL THREE;

        E;

        T "EMU" = 6;
```

(Decisions, cont.)

```
6     M;
      W "DIA" AND "VES" ARE CORRECT.  THE THIRD ONE IS "SIL";
      E;
      T "EMU" = 5;
7     M;
      W "SIL" AND "VES" ARE CORRECT.  THE THIRD ONE IS "DIA";
      E;
      T "EMU" = 3;
8     M;
      W "SIL" AND "DIA" ARE CORRECT.  THE THIRD ONE IS "VES";
      E;
      T "EMU" = 1;
9     M;
      W "SIL" IS RIGHT.  THE OTHER TWO ARE "DIA" AND "VES";
      E;
      T "EMU" = 2
10    M;
      W "DIA" IS RIGHT.  THE OTHER TWO ARE "SIL" AND "VES";
      E;
      T "EMU" = 4;
```

(decisions, cont.)

```
11    M;

      W "VES" IS RIGHT.  THE OTHER TWO ARE "SIL" AND "DIA";

      E;

      J BAR;

      E;

      W ONE OF THEM ---etc--- TRY AGAIN.;

      J HAWKEN;

BAR   ========
```

The above routine combines effective control with economy of means, besides lending itself to further extension. Note that it enabled us to improve on the earlier "shotgun" response to answers containing only two out of the three terms sought.